



GammaWare^[Gw] User's Guide

Version 3 - from 10/2009 to ...

AGATA Data Analysis Team¹

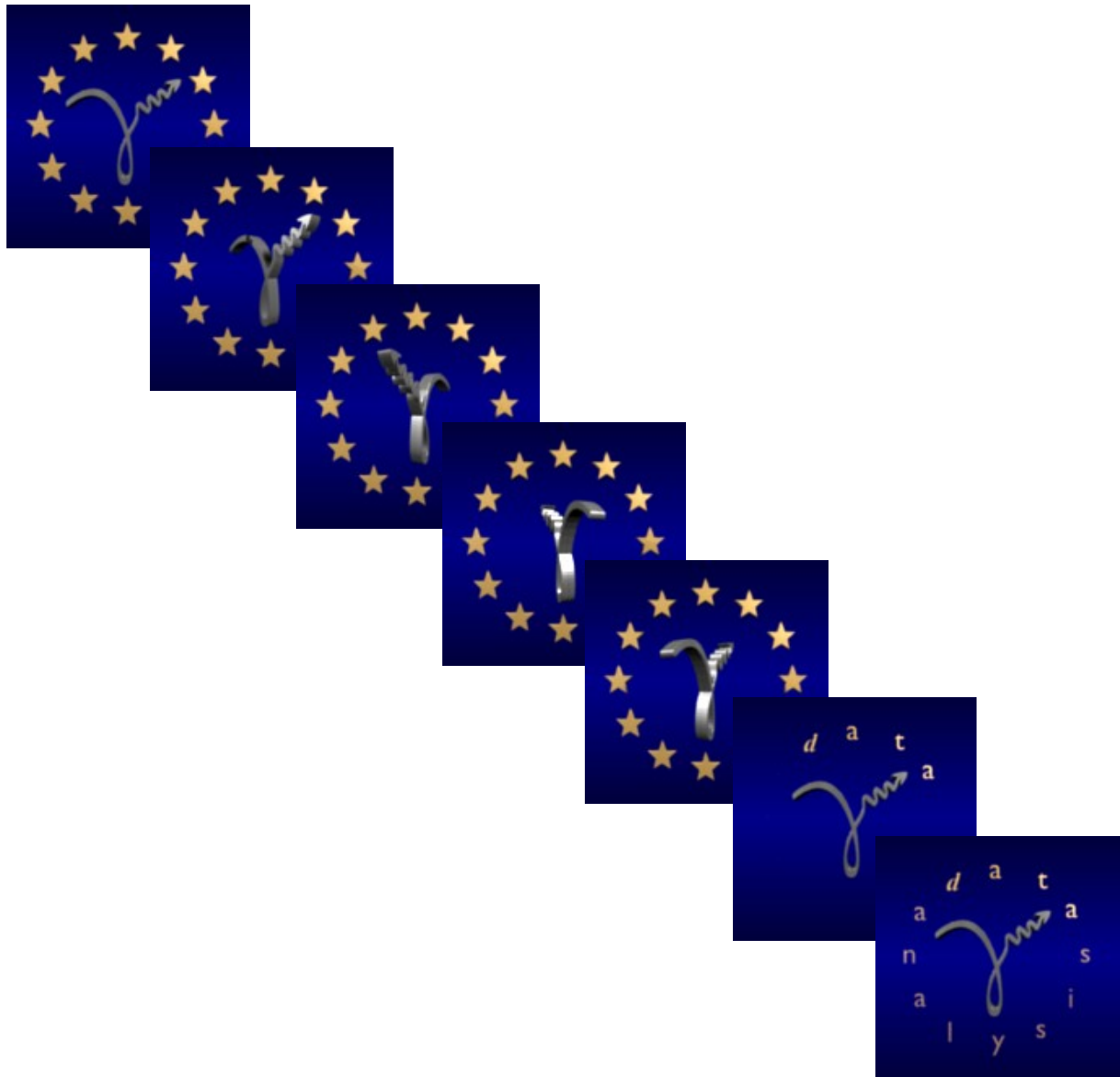
This document gives an overview of the GammaWare [Gw] package which has been developed within the AGATA collaboration in particular to analyse AGATA like data. It could be however used in a more general purpose for discrete gamma-ray spectroscopy.

¹ To get this document and more informations, go and see <http://agata.in2p3.fr>

Contents

Chapter 1 - Introduction.....	3
Chapter 2 - Code organisation and documentation.....	6
2.1 Version control.....	7
2.1.1 General considerations.....	7
2.2.2 GammaWare repository.....	9
2.2 Source documentation.....	11
2.3 Web log and syndication.....	13
2.4 Bug tracker.....	14
2.5 Mind maps.....	15
2.6 Users' guide.....	15
2.7 Video tutorials	16
Chapter 3 - Requirements and installation.....	17
3.1 The GammaWare package.....	18
3.2 Other packages.....	21
Chapter 4 - The GammaWare in details.....	22
4.1 Overview.....	23
4.2 The Core component	24
4.2.1 The Log System.....	24
4.3 The Physics component.....	24
4.3.1 Links, Level and Level Scheme.....	24
4.3.2 Level Scheme Player.....	24
4.4 The GEM component.....	24
4.5 The Tools component.....	24
4.5.1 Conversion of histograms.....	24
4.5.2 Spectrum Player.....	24
4.6 The ADF component	25
4.6.1 Introduction.....	25
4.6.2 Keys and Frames.....	26
4.6.2.1 Keys.....	27
4.6.2.2 Frames.....	29
4.6.3 Interface to Narval.....	34
4.6.4 Interaction of the algorithms with the Data Flow	36
4.7 The AD FE component.....	40
4.7.1 Watchers and FrameDispatcher.....	40
Chapter 5 - Examples	42
5.1 Some macros in details.....	43
5.2 Some demos in details.....	43
5.2.1 demos/adf.....	43
5.2.1.1 Watchers.....	43
5.2.2 demos/gem.....	46
5.2.3 demos/tools.....	47
Glossary.....	50
Annex A : summary form for some actors	51

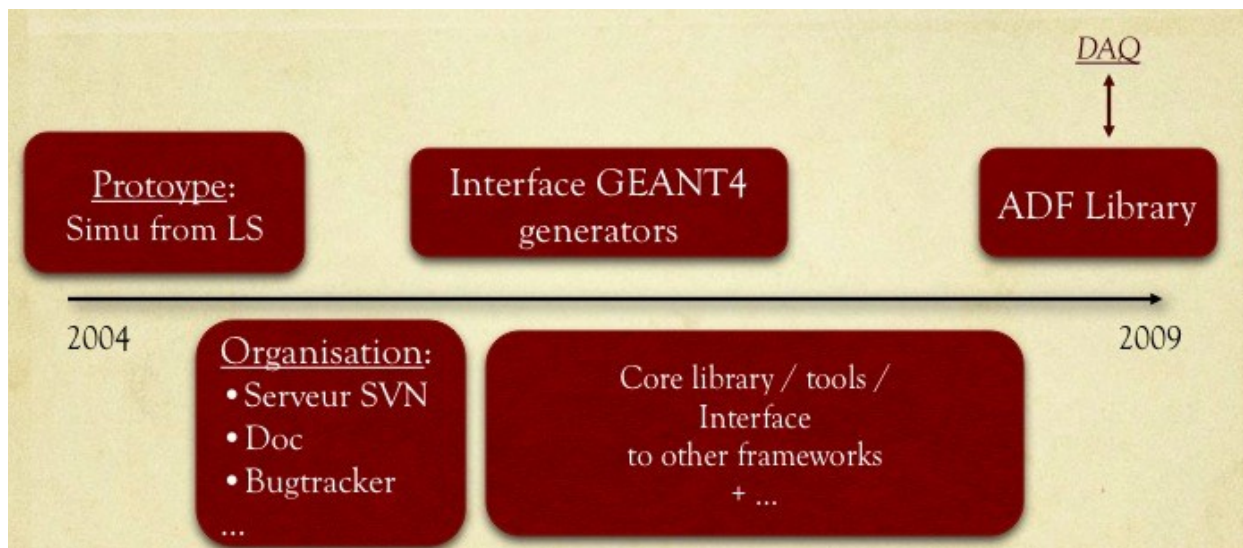
Chapter 1 - Introduction



Chapter 1 Introduction.....	3
-----------------------------	---

The GammaWare package (GW) is developed for the AGATA community and thus provides facilities, tools to help for analysis of data coming out from the AGATA detector (real or simulations from GEANT4). Following the prescriptions of the first meeting (NBI June 12-13 2003), the official programming language is C/C++ on a Linux platform. We do emphasize that FORTRAN functions may be easily linked in C/C++ programs. Other language might be used in the future (python for script, Java for GUI ...), but the core of the framework is C/C++. Concerning the platform, particular attention is devoted to have as much as possible a multi-platform package even if Linux is the reference.

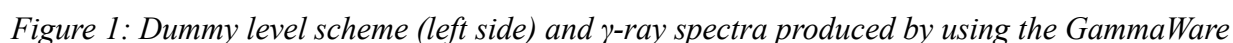
For all these reasons, this package is widely based on the ROOT <http://root.cern.ch> framework which is written in C++. ROOT already defines many objects that are currently used by physicists: histograms, graphs, functions, cuts, detector geometries, etc All these objects can be stored in ROOT files. A large number of methods comes with those objects to manipulate them. For instance to fit histograms, display them or to search for peaks. ROOT also proposes a solution to store events in an efficient tree-like structure with some facilities to analyse them.



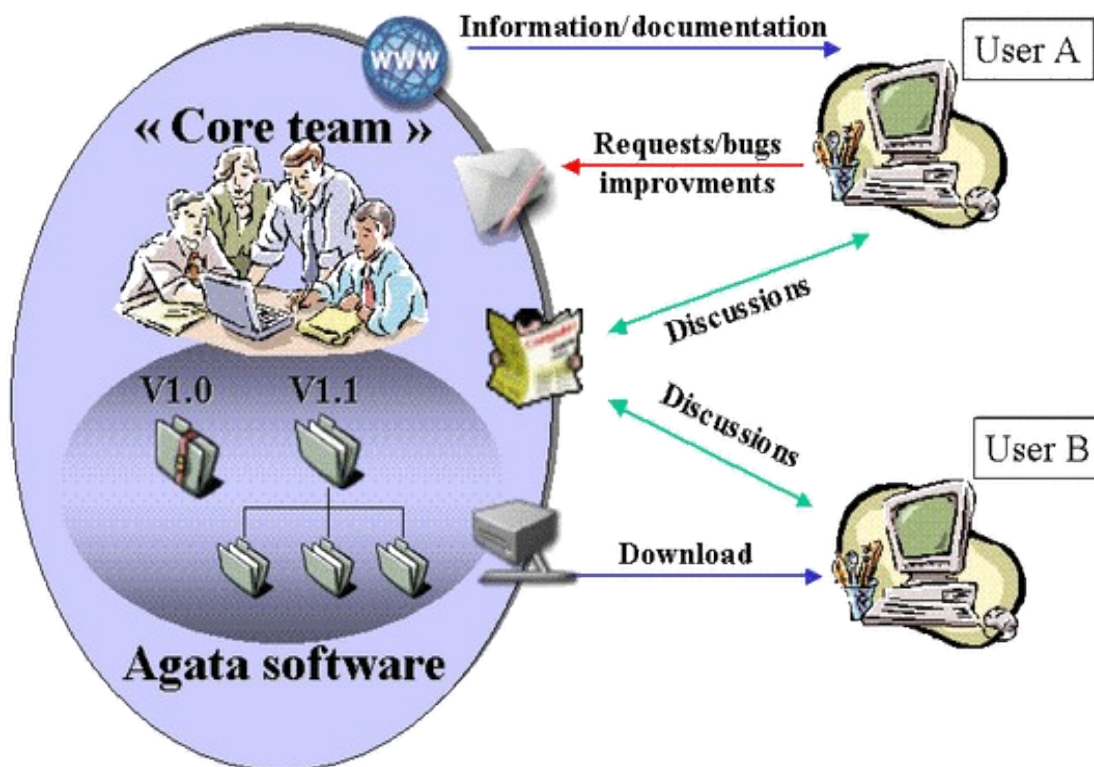
Many objects defined in the GammaWare package are embedded in ROOT so that they can be stored in ROOT files, displayed using ROOT GUI. It is the case for some objects particular to γ -rays spectroscopy: nuclear levels, γ -rays or level schemes. Associated with these objects comes methods, algorithms to play with them.

The GammaWare project has started in 2004 and, at this time, consisted in a single library. The goal was to produce randomly cascades from a **LeveScheme** (based on their relative intensities) so that it can be used as an input for GEANT4 simulations. This is illustrated in Figure 1 where are displayed a dummy and simple level scheme (on the left side) together with some spectra (on the right side) generated by the GammaWare from it. In the first spectrum, all the produced γ -rays cascades are present. The other two spectra show gated spectra from which one can see the consistency between the spectra and the cascade selected by the two different single gates. The package has grown over the years to include more and more features like a library to interact with the AGATA Data Flow, to plug tracking and PSA algorithms, to build graphically a LevelScheme and use it gate on high order correlated space.

Figure 1 is an energy level diagram for the ^{238}U nucleus. The vertical axis represents the excitation energy in MeV, ranging from 0 to 6. The diagram shows several decay chains. A red box highlights the 573 MeV level ($45/2^+$) and the 53/2+ level above it. A blue box highlights the 656 MeV level ($29/2^+$) and the 25/2+ level below it. Arrows indicate transitions between levels, with some labeled with half-lives like 754, 417.9, and 712.



Chapter 2 - Code organisation and documentation



Chapter 2 - Code organisation and documentation.....	7
2.1 Version control.....	9
2.1.1 General considerations.....	9
2.2.2 GammaWare repository.....	11
2.2 Source documentation.....	17
2.3 Web log and syndication.....	18
2.4 Bug tracker.....	19
2.5 Mind maps.....	20
2.6 Users' guide.....	21
2.7 Video tutorials	21

The package is maintained/developed by several persons at the same time with (hopefully) many users working with it. It is then important to organise this collaborative development and also crucial to define how the different actors speak together. The solutions, presented in this chapter, could be found on the official Agata Data Analysis web site <http://agata.in2p3.fr>.

The developers work on the same files. So, to avoid conflict when a file is modified, the package is under **version control**. The adopted solution is presented in the first section of this chapter. When writing code, comments are important to know exactly what is doing a particular function, to understand a tricky algorithm etc... . The second section describes how the comments could be used to generate automatically an **HTML documentation of the package**. To be informed of important results, a **web log** is used to publish notes on the web and several **rss feeds** are available (see section 2.3). Bugs are part of any program. What is important is to find them, correct them so that the code will become more and more reliable over the year. In order to allow the user to report for a bug, to ask for new features and to follow how is treated the issue, a **bug tracker** is available. The details concerning the issue tracking are presented in the Bug tracker section.

2.1 Version control

2.1.1 General considerations

The source files are shared by different person working in different institutes. Obviously it would be a mess if all of them have their own copy of the package. Thus, the files are under version control. Two solutions exist in the freeware world: CVS and SVN also known as subversion. CVS is probably the most popular one but also the older one. SVN is more recent and proposes more flexibility compared to CVS. So the choose has been to work with a SVN <http://subversion.tigris.org/> server. With subversion, the developers can share the same files so that if two of them submit modifications on the same file, the system detects possible conflicts and update the repository only if there are any or when all of them have been solved (automatically or manually).

Subversion is a free/open source version control system. The central point is called a repository. The repository is like a file server that remembers every changes made to files and directories. This allows you to recover old versions of files or examine the history of how files have changed. SVN can access a repository (equivalent to a directory) across networks which allows files to be shared/used/modified by people working on different computers. At the beginning, the revision number is 0 (R0). Any successful submission to the repository will increment this number by 1. The typical work cycle with subversion is illustrated on the following picture :

- If you don't have yet the sources, you must get them from the repository. This is done in the picture with: *svn checkout file:///path/to/repo/gw/trunk gammaware* .
- you have now a copy on your own computer. You can have some informations about the package *svn info* or *svn log*. For this tutorial, let says the revision number is 0.
- You can now modify any files. Try it and check out your changes with: *svn status*. Remember: you are changing the local copy, not the file on the repository !! so there is no need to worry about loosing something. If you want to add/delete new files/directory, you'd better used: *svn add*, *svn delete*, *svn copy*, *svn move*, *svn mkdir* ...

- Once you are satisfied with your modifications or you don't want to loose what you've just done, you are ready to send the file (or all the files) to the repository. However, somebody else may have already submit its modifications (on the same file or another one), as illustrated on the picture by R1 (revision 1) or R2 (revision 2). Before sending your modifications, be sure there is no conflict with other's modifications. To check this, run: *svn update* if the letter c does not appear on the first column, you can commit your changes with: *svn commit -m 'the comment about the modification' ; svn update*
- Restart the cycle ... In case of problems (conflicts), svn gives you tools to help you resolving the conflicts: *svn merge*, *svn revert*, *svn resolved* ...

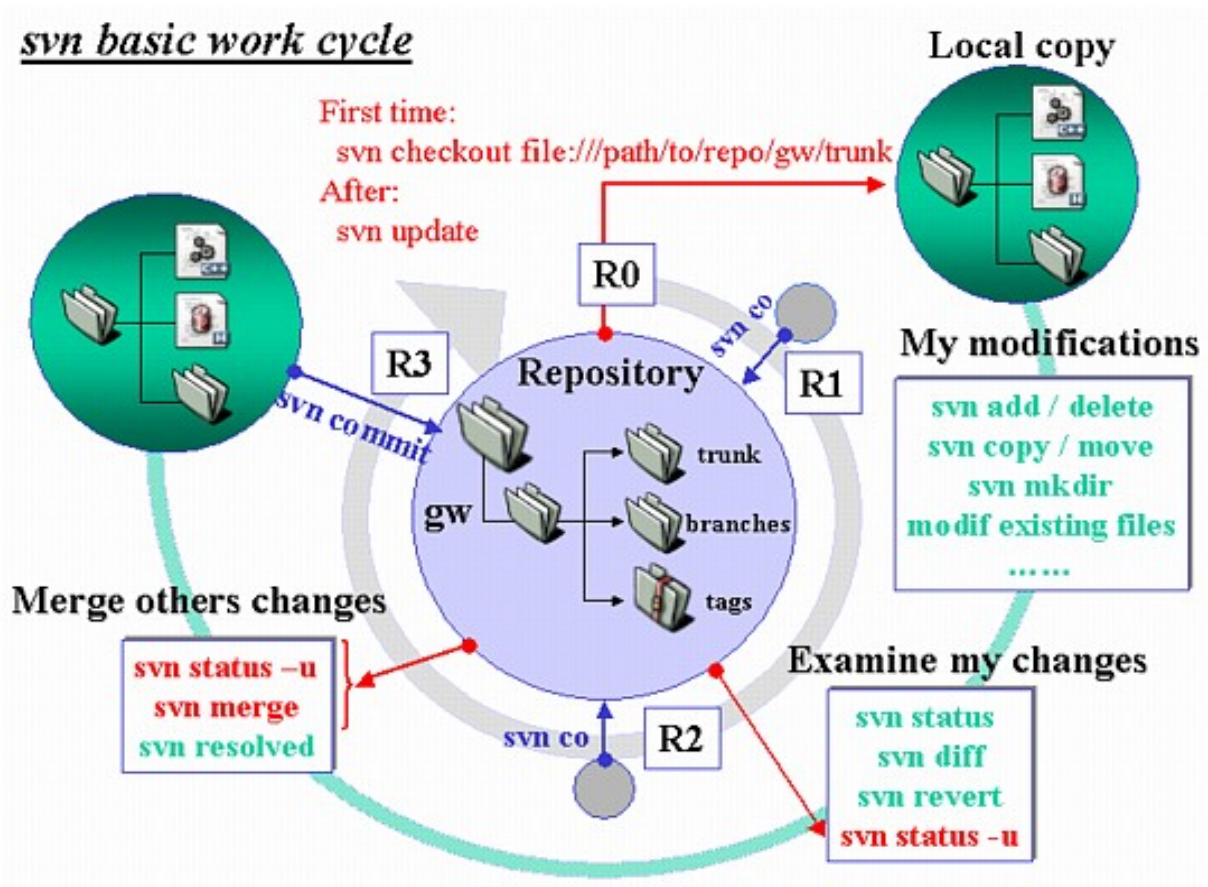


Figure 2: Typical work cycle with subversion

Because of the version control, the GW package is available to the user with version numbers. The HEAD version corresponds to the very last modifications. As a consequence, this version may come with bugs. Any previous version is available any time. Once major developments are accomplished, the code tested without bugs, a tag version is produced which corresponds to a “snapshot” of the project in time. So, if you want to use the very last features, you'd better work with the HEAD version but if you want to use stable versions, the tag versions are the ones to be used.

2.2.2 GammaWare repository

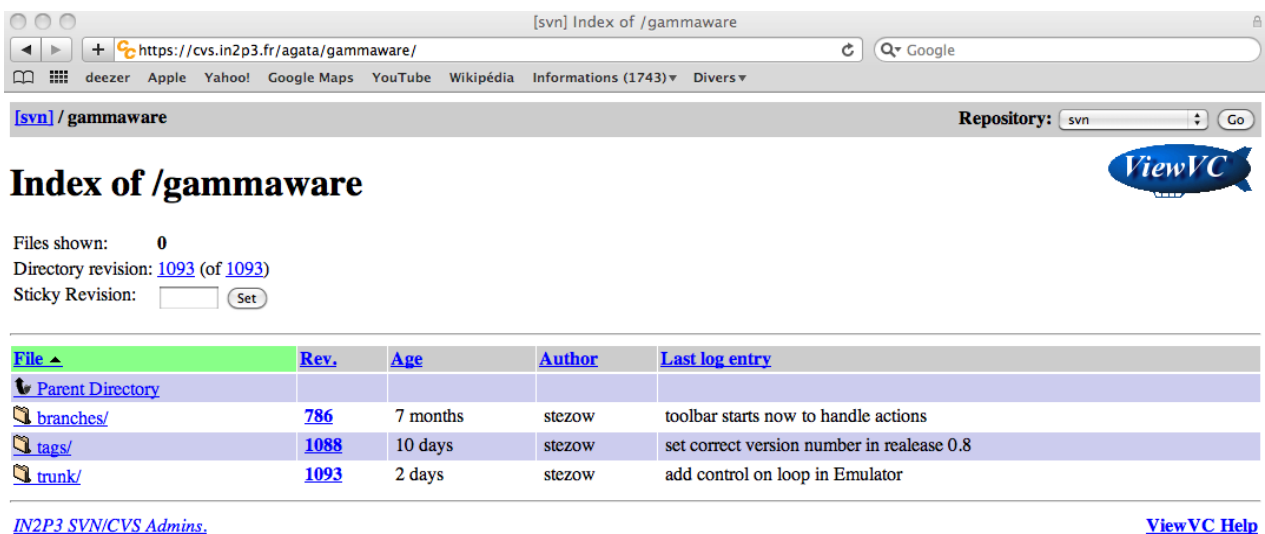
The GammaWare repository is divided in three directories.

- trunk : it is the main development line
- tags : it is the directory where the released versions are stored
- branches : a branch is created to start important developments (consequent modifications of the interface) without interfering with the main line or to start new projects.

The svn repository is hosted at the CCIN2P3. It could be browsed with an interface that shows the modifications of any single file at the following web address :

<https://svn.in2p3.fr/agata/gammaware>

This is a secure web site protected by an username and a password. To get it, please send an email at agata(AT)ipnl.in2p3.fr. The entrance window (see Fig 5) gives the current revision number, in this case 1093, shows the different directories and the last version for which a file has been modified. The name of the operator is given together with a sentence that describes the corresponding modifications.



The screenshot shows a web browser window with the address bar displaying `https://cvs.in2p3.fr/agata/gammaware/`. The page title is "[svn] Index of /gammaware". Below the address bar, there is a navigation bar with links to various services like deezer, Apple, Yahoo!, Google Maps, YouTube, Wikipédia, Informations (1743), and Divers. The main content area is titled "Index of /gammaware" and includes a "Repository:" dropdown set to "svn" and a "Go" button. A "ViewVC" logo is also present. The interface shows the current directory revision as 1093 (of 1093) and a "Sticky Revision:" field. Below this, a table lists the files and directories in the repository:

File	Rev.	Age	Author	Last log entry
Parent Directory				
branches/	786	7 months	stezow	toolbar starts now to handle actions
tags/	1088	10 days	stezow	set correct version number in realease 0.8
trunk/	1093	2 days	stezow	add control on loop in Emulator

At the bottom of the page, there are links for "IN2P3 SVN/ CVS Admins." and "ViewVC Help".

Figure 3: The GammaWare is composed of three directories branches, tags and trunk

The interface is fully browsable and the history of any single file could be tracked back. As well is it possible to display the modifications that have been done from one version to another one as illustrated in Illustration .

The svn server could be accessed by anybody using the `svn+ssh` protocol. This could be done though an anonymous account in reading mode only using the command :

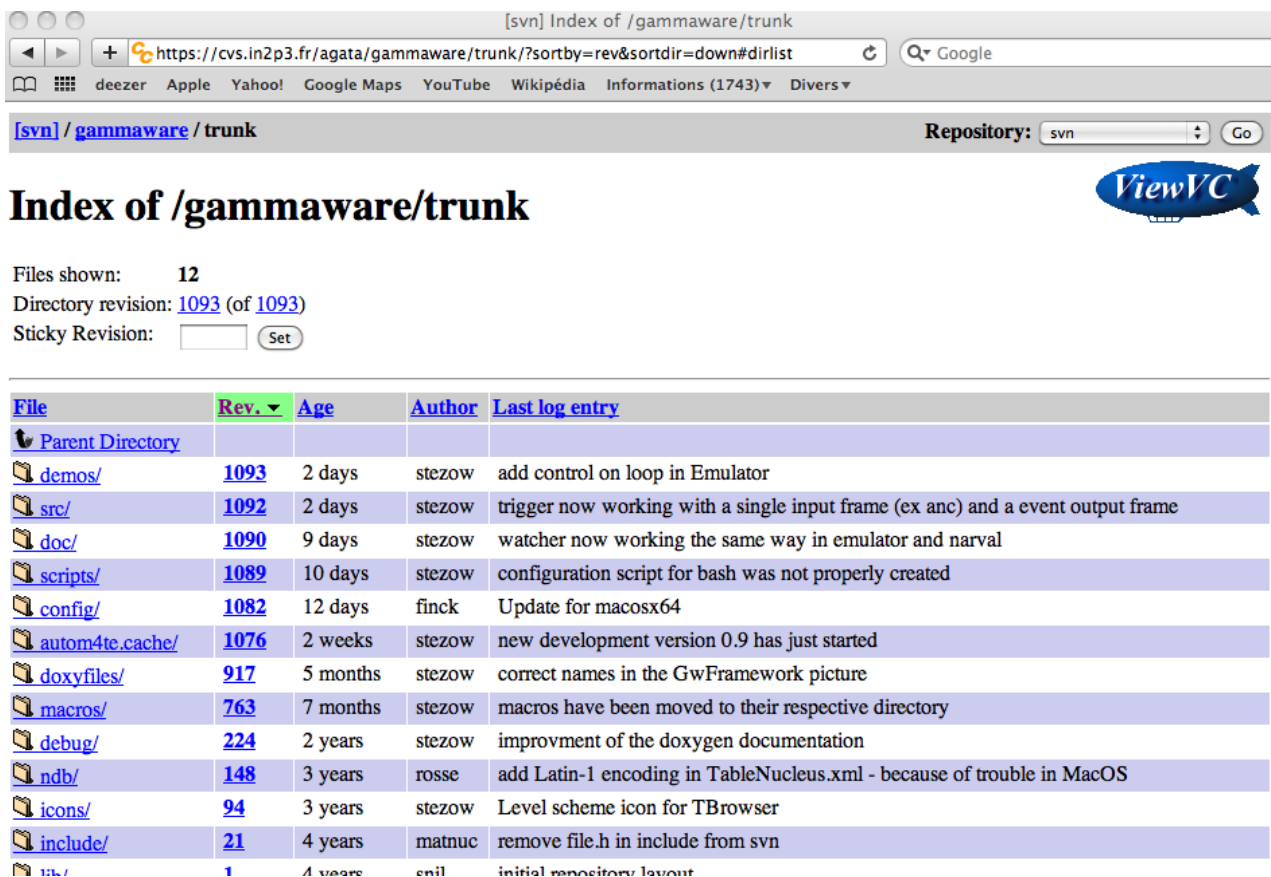
```
svn co svn+ssh://anonsvn@anonsvn.in2p3.fr/agata/gammaware/trunk gammaware
```

In this case, it downloads the very last version (head version). You may need to set up properly your

ssh configuration (see chapter 3 for more details). If you would like to be part of the developers and be able to modify the GammaWare package, you need to access the repository in reading and writing mode. For that, you should generate your own keys. Use the command:

```
ssh-keygen -t rsa -b 2048 -f ~/.ssh/key_svn_in2p3_MYLOGIN
```

where MYLOGIN is the login name you will have to reach the repository. It will generate two files in your .ssh directory. One with the name `key_svn_in2p3_MYLOGIN`, the other one with the name `key_svn_in2p3_MYLOGIN.pub` that represent respectively the private key and the public key. The private key stays on your machine and MUST NOT BE DISTRIBUTED. It is the guarantee for a secure connection.



Files shown: 12
Directory revision: 1093 (of 1093)
Sticky Revision: Set

File	Rev.	Age	Author	Last log entry
Parent Directory				
demos/	1093	2 days	stezow	add control on loop in Emulator
src/	1092	2 days	stezow	trigger now working with a single input frame (ex anc) and a event output frame
doc/	1090	9 days	stezow	watcher now working the same way in emulator and narval
scripts/	1089	10 days	stezow	configuration script for bash was not properly created
config/	1082	12 days	finck	Update for macosx64
autom4te.cache/	1076	2 weeks	stezow	new development version 0.9 has just started
doxyfiles/	917	5 months	stezow	correct names in the GwFramework picture
macros/	763	7 months	stezow	macros have been moved to their respective directory
debug/	224	2 years	stezow	improvement of the doxygen documentation
ndb/	148	3 years	rosse	add Latin-1 encoding in TableNucleus.xml - because of trouble in MacOS
icons/	94	3 years	stezow	Level scheme icon for TBrowser
include/	21	4 years	matnuc	remove file.h in include from svn
1.1.1/	1	4 years	enil	initial repository layout

Figure 4: The complete history of the modifications of any file is kept

The public key is the one that can be distributed. Send it at `agata(AT)ipnl.in2p3.fr` together with the name of your laboratory. Once it is done your public key will be added on the svn server and you will have access to the repository from the command line. In order to not give your passphrase each time you have an action on the repository, you'd better keep your key in memory. For that, run the command:

```
ssh-add ~/.ssh/key_svn_in2p3_MYLOGIN
```

Your passphrase will be asked, and, in case of success, the key will be kept in memory (try for instance `ssh-add -l`) and, if you run the command

```
svn checkout svn+ssh://MYLOGIN@svn.in2p3.fr/agata/gammaware/trunk gammaware
```

you will see the passphrase is not asked anymore and you will have a local copy of gammaware on your computer. Depending on *ssh* configurations, you may need to add the following lines in the *.ssh/config* file:

```
Host svn.in2p3.fr
  User MYLOGIN
  PasswordAuthentication no
  IdentityFile ~/.ssh/key_svn_in2p3_MYLOGIN
  RSAAuthentication yes
  PubkeyAuthentication yes
  Protocol 2
  ForwardX11 no
  ForwardAgent no
```

The screenshot shows a web browser window displaying a diff of the file `/gammaware/trunk/src/adf/NarvalInterface.cpp` between two revisions. The browser's address bar shows the URL `https://cvs.in2p3.fr/agata/gammaware/trunk/src/adf/NarvalInterface.cpp?sortdir=dow`. The page title is "Diff of /gammaware/trunk/src/adf/NarvalInterface.cpp". The diff is presented in a table-like format with two columns: "revision 1087, Wed Nov 4 17:54:08 2009 UTC" and "revision 1092, Fri Nov 13 17:54:06 2009 UTC". The left column shows the original code, and the right column shows the modified code. The diff highlights changes in the configuration file path handling logic. A legend at the bottom indicates that red lines represent "Removed from v.1087", yellow lines represent "changed lines", and green lines represent "Added in v.1092".

#	Line 256	Line 256
256	// now try to open an ADF configuration file (if required) and	// now try to open an ADF configuration file (if required) and
257	// configure/reconfigure (or add information) to the global agent	// configure/reconfigure (or add information) to the global agent
258	switch(do_adf_conf){	switch(do_adf_conf){
259		case 0:
260		break;
261	case 1:	case 1:
262	if (! ::getenv("ADF_CONF_PATH")){	if (! ::getenv("ADF_CONF_PATH")){
263	tmp += "ADF.conf";	tmp += "ADF.conf";

Legend:
Removed from v.1087
changed lines
Added in v.1092

Figure 5: Modifications from one version to the next one could be displayed

2.2 Source documentation

When writing softwares, it is necessary to comment the code itself. For the developer first because he may not be able to understand himself what he has done six month after having writing a piece of code (tricky algorithm, functions not yet implemented or not completely ...). Because the code is shared, another guy may have some troubles to understand another logic. As well the user must know the purpose of a class, a function, the possibilities/restrictions when using them. Browsing the code itself may become quickly a nightmare because files are in different directories.

A widespread solution, based on *Doxygen* (<http://www.stack.nl/~dimitri/doxygen/>), is used to produce an HTML documentation from the source files. Because *Doxygen* parses the source files, the comments in the code must be included at a precise position with a given format (the constraints are however loose). The produced documentation could be browsed on the Agata Data Analysis web site <http://agata.in2p3.fr> (documentation section) only for the development version. One of these pages is displayed in figure 6.

On the left of the page, a menu helps you to navigate through different categories while on the top a **Quick search** is provided to find a key word in the whole documentation. A large part of the window is used to display the results. Here is a non-exhaustive list concerning the menu.

Main - Page gives a short introduction about the project

File List is the list of all the files with a brief description about the content

Class List is the list of all the classes (or structs, unions and interfaces) with a brief description

Class Hierarchy displays the inheritance tree (textual or graphical)

Class Members displays the list of all class members with links to the classes they belong to

Namespace List displays the list of all namespaces with brief descriptions

Directories displays the directory hierarchy sorted roughly, but not completely, alphabetically

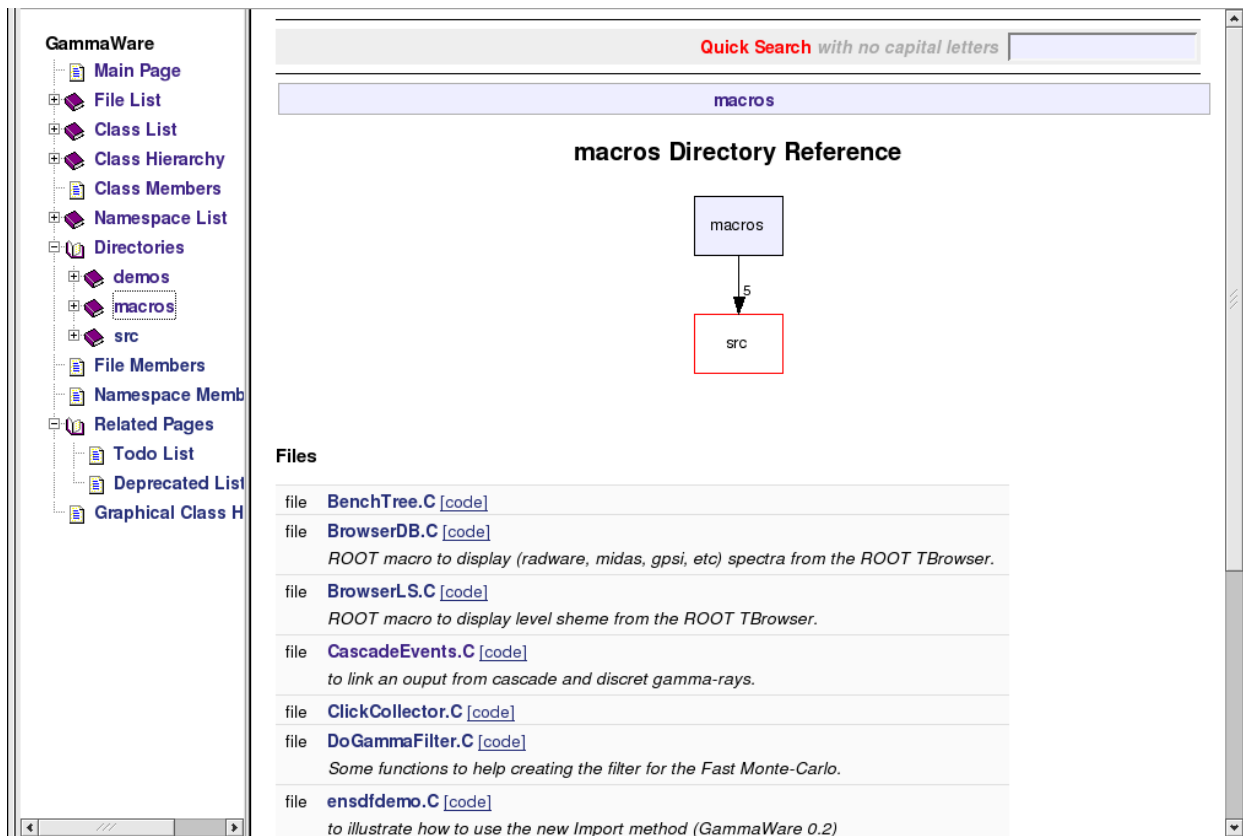


Figure 6: Doxygen web page associated with the macros directory

The picture 5 corresponds to the informations concerning the Directory *macros*. It shows all the *macros* that are defined in the package (BenchTree.C, BrowserDB.C etc ...) as well as a brief description (if available) on their purpose. By clicking on the name of the file, you will get more informations about the functions that are defined in each macro. The code itself is also available just

by clicking on the right [code] link. As it can be seen, there are many different ways to get the information about a class, a method, a function ... if the developer has introduced the proper comments. The documentation is updated every night and thus corresponds to the head version. If you would like more informations, please don't hesitate to send an email [agata{AT}ipnl.in2p3.fr] or post an issue on the bug tracker.

2.3 Web log and syndication

In order to inform the user of important results during the different stages of the project, a *web log* has been set up to publish short notes. This *web log* could be consulted through the official Agata Data Analysis web site <http://agata.in2p3.fr> (Weblogs section). A snapshot of this web log is displayed in the figure 7.



Figure 7: Snapshot of the Agata Data Analysis Web log

On the right side of the page several possibilities exist in order to search for a note by using the calendar, by key words or by categories (General, Tips, meeting's minutes etc ...). The notes themselves are displayed on the left side. Possibilities are given to the user to add a comment for each individual note (a form appears at the end of the note or by clicking on the no comments link). It can be used for instance if the user has additional informations to add, to ask for more informations about the results published in the note.

Because notes are published now and then, it is important for the user to be informed of new inputs without having to browse the web site. For that purpose, *rss feeds* are provided. A *rss feeds* is a file (written using the xml language) that is updated each time a new note is added to the web log. To be

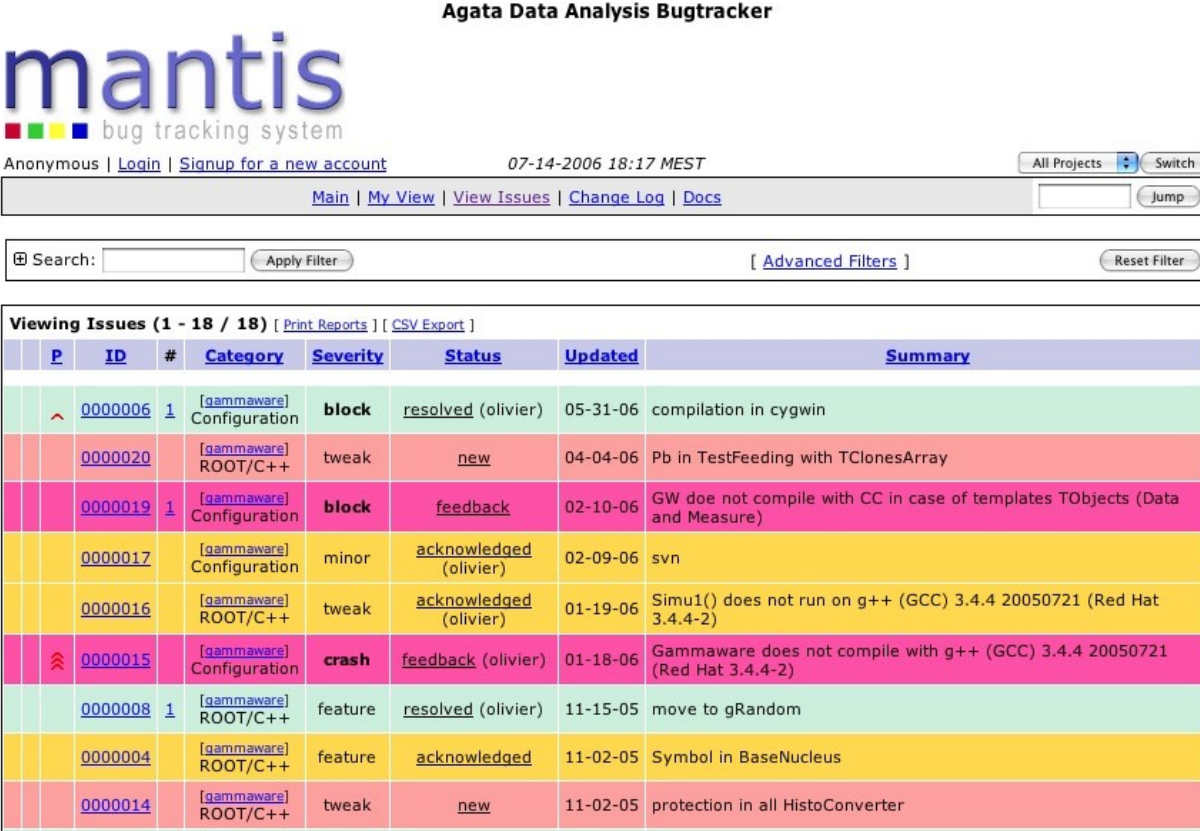
informed of any modification, the user must use a *rss reader* which checks any modifications of the xml file and notify new entries. One *rss reader* could be RSSOwl (Java based application) as explained in one of the note of the web log “being informed of any new entry in the logbook”. Of course there many different possibilities (thunderbird, firfox) and you can choose the one you prefer.

Other *rss feeds* are also provided on the Agata Data Analysis web site <http://agata.in2p3.fr> to inform the user of a new version of the GammaWare package (development version) and some are related to the bug tracker (see next section). A discussion forum is also available to exchange informations/tips at the following address: <https://www.agata.org/elog/agata/>.

2.4 Bug tracker

When using the code, the user (or a developer) may discover a bug, may need a new feature or some changes. It would not be very efficient just to send private emails. All the informations are joined together in a *bug tracker* that can be consulted from the Agata Data Analysis web site <http://agata.in2p3.fr> (Bugtrackers section). Because the GammaWare depends strongly on ROOT, a link to the ROOT bug tracking system is also provided. A snapshot is shown in figure 8.

Agata Data Analysis Bugtracker



P	ID	#	Category	Severity	Status	Updated	Summary
	0000006	1	[gammaware] Configuration	block	resolved (olivier)	05-31-06	compilation in cygwin
	0000020		[gammaware] ROOT/C++	tweak	new	04-04-06	Pb in TestFeeding with TClonesArray
	0000019	1	[gammaware] Configuration	block	feedback	02-10-06	GW doe not compile with CC in case of templates TObjects (Data and Measure)
	0000017		[gammaware] Configuration	minor	acknowledged (olivier)	02-09-06	svn
	0000016		[gammaware] ROOT/C++	tweak	acknowledged (olivier)	01-19-06	Simu1() does not run on g++ (GCC) 3.4.4 20050721 (Red Hat 3.4.4-2)
	0000015		[gammaware] Configuration	crash	feedback (olivier)	01-18-06	Gammaware does not compile with g++ (GCC) 3.4.4 20050721 (Red Hat 3.4.4-2)
	0000008	1	[gammaware] ROOT/C++	feature	resolved (olivier)	11-15-05	move to gRandom
	0000004		[gammaware] ROOT/C++	feature	acknowledged	11-02-05	Symbol in BaseNucleus
	0000014		[gammaware] ROOT/C++	tweak	new	11-02-05	protection in all HistoConverter

Figure 8: Snapshot of the Agata Data Analysis bug tracker

On the first page is displayed some news concerning the project. As for the *web log*, *rss feeds* are provided so that it is not necessary to browse the web pages to be informed of something new. The “View Issues” gives something similar to picture 8 where several bugs are reported. The colours

inform about the status of the issues (new, acknowledge, resolved, closed etc ...). For the moment, the session's login is anonymous so that the issues can only be browsed. To report for a new issue, the user must “Login”. The first time, just click on the “Sign up for a new account”. After that, fill the form to submit a new bug. It is then possible to follow any modifications concerning your request and the others know if the bug they have just discovered has already been reported.

2.5 Mind maps

To follow the current developments, mind maps are provided in the distribution. As for the *doxygen* documentation, the mind maps could be browsed on the AGATA Data Analysis web site <http://agata.in2p3.fr> (menu Recent news). There are also updated every nights. Figure 9 shows such a map. The different sub-projects are there with the foreseen developments. In this case a bug in the Log Level has been corrected and an additional method (switch_output_block) is going to be implemented soon. The content of a mind map depends on the developer ... so it may not give an exhaustive overview of the current developments. In any cases, go and browse the GammaWare repository to check any single modifications.

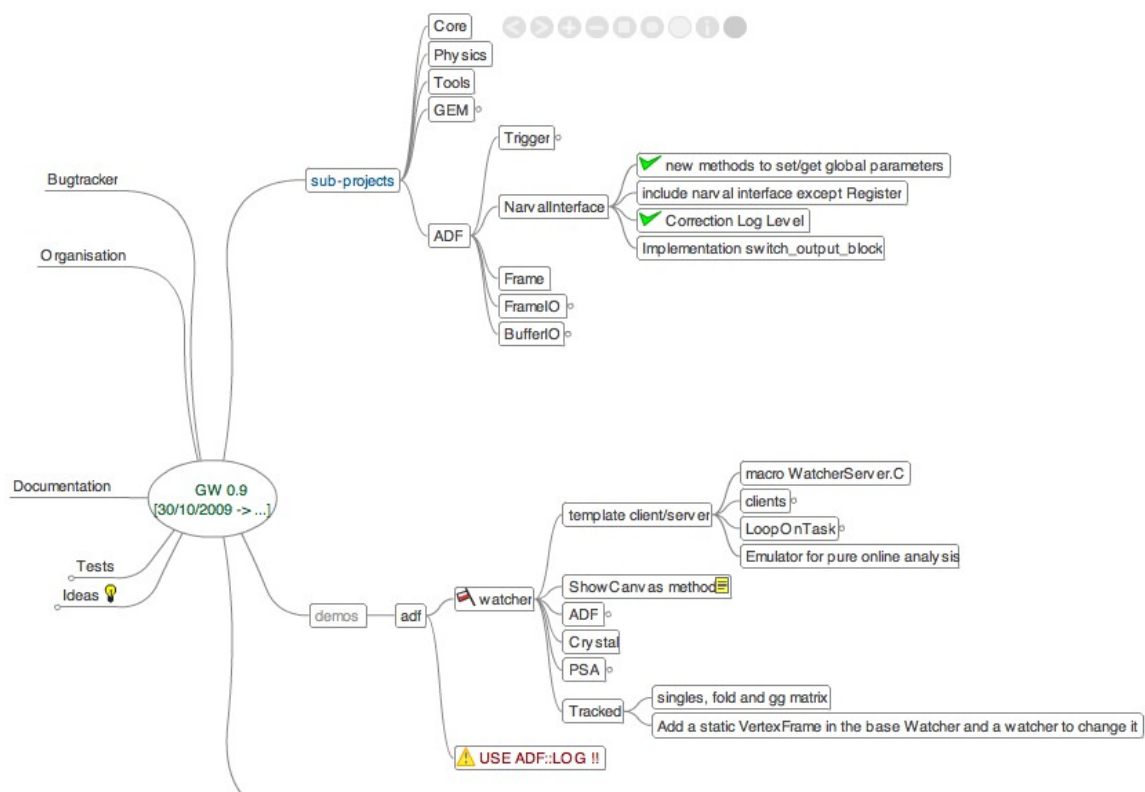


Figure 9: The mind maps show current developments

2.6 Users' guide

To be written

2.7 Video tutorials

Some video tutorials are available. They are made with quicktime and available at the following address:

<http://agata.in2p3.fr/VT/>

You cannot browse directly the directory. So to get a file named Avideo.mov, just point to the complete address i.e. <http://agata.in2p3.fr/VT/Avideo.mov> in your html browser or in quicktime or download the file to watch it with your favourite video reader.

Here is a list of some video tutorials available:

- Watchers_Default_Online_Offline.mov shows how to start online and offline watchers.
 - Date : < 11/11/2011. Very first version of watchers
- Gw_Fit.mp4 shows how to fit graphically peaks in a root canvas
 - Date : 04/2011.
- OnlineGLP_1.mp4 shows how to :
 - Date : 07/2011
 - play with online watchers (Global Level Processing) - start, stop, set active watchers
 - display some basic spectra, reset them, tag some spectra though the task menu
 - display any spectra produced the way you want
 - save all spectra produced in root files (with/without automatic calls)
- LS_2.mov shows how to :
 - Date : 11/2011
 - build graphically a level scheme
- CS_1.mov shows how to :
 - Date : 11/2011
 - play graphically with a level scheme and put gates on a correlated space (2D matrix)

Chapter 3 - Requirements and installation



Chapter 3 - Requirements and installation.....	17
3.1 The GammaWare package.....	19
3.2 Other packages.....	23

3.1 The GammaWare package

As already said, the GW package is build on the top of ROOT. Thus, the basic requirement is to have it installed on your system. In order to have a coherence between the two frameworks, each version of the GammaWare is developed using a unique version of ROOT. Here is a non-exhaustive list :

Gw	ROOT
0.1	5.10/00
0.2	5.10/00
0.4	5.10/00
0.5	5.20/00
0.6	5.20/00
0.7	5.20/00
0.8	5.20/00
0.9	5.24/00b

This list may not be up to date, check the .GWVERSIONS file in the top directory of the GammaWare package to have the last version. Even numbers are stable version (tags/releases) for which the development has been frozen. There are two ways to get the GammaWare package :

As a tarball.

There are tarballs for all the released versions with the following form

GammaWareX.Y.tar.gz

As well a development version

GammaWare-dev.tar.gz

is produced automatically every nights together with an html documentation of the code which could be browsed there:

<http://www.ipnl.in2p3.fr/gammaware/doc/html/>

Unzip and untar this file with following command:

```
gunzip GammaWare-dev.tar.gz
tar xvf GammaWare-dev.tar
```

From the svn server.

An anonymous access is available (reading mode only) to get the released versions and the last development version. You can of course use your own access to the svn if you have one (see ch, section). To download the package from the svn server:

svn co svn+ssh://anonsvn@anonsvn.in2p3.fr/agata/gammaware/trunk gwdev

or for released versions

svn co svn+ssh://anonsvn@anonsvn.in2p3.fr/agata/gammaware/tags/release-X.Y gwX.Y

Before running these commands you must configure your ssh client with:

For OpenSSH configuration in your ~/.ssh/config file, add these lines:

```
Host anonsvn.in2p3.fr
  Port 2222
  User anonsvn
  PasswordAuthentication yes
  RSAAuthentication no
```

```
PubkeyAuthentication no
ForwardX11 no
ForwardAgent no
```

From "SSH.Com" (or F-Secure) SSH configuration.

The main configuration file is `~/.ssh2/ssh2_config`, to which a specific section must be added:

```
anonsvn.in2p3.fr:
  User anonsvn
  Port 2222
  AuthenticationSuccessMsg no
  AllowedAuthentications password
```

Whatever the method, you should have, in the directory where you download the GammaWare, a new directory which contains the entire package. Go to this directory.

The first step consists in running the script called `configure`. As it can be guessed, this script checks if everything is OK to build the package and, in case of success, it creates the makefile that are used to effectively build the GammaWare. One of the most important check consists in looking if ROOT is properly installed on your machine. If not, you will get the following message:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
The root-config script is missing !
root-config is mandatory to build this package.
Check out if ROOT is installed on this machine.
ROOT is available here [http://root.cern.ch/].
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

And, in case of success:

```
-----
Check the installation parameters with:
make showconf
To build the gammaware package:
make
and to install the package:
make install
More informations about the configuration are available in README
In case of problems, please report <agata@ipnl.in2p3.fr>
-----
```

Several options are available for the `configure` script. To know more about them, read the README and INSTALL files which are located at the top of the GammaWare directory. One of the most important one is the `--prefix` option which determines where will be installed the package. The default value is `/usr/local`. If you are not administrator on your machine (root user), you will not be allowed to write anything in this directory. So to select another location to build the package, you should use the `--prefix` option:

```
configure --prefix=/where/i/want/my/package
```

In fact it is strongly recommended to install the package in another directory and not in the

GammaWare directory itself since any uninstall procedure is not yet implemented. Once the configure script has run correctly, you can check the configuration with:

```
make showconf
```

and you will get some informations on the package itself and the installation destination. Then to build the package, just run:

```
make
```

You will get messages showing the different sub-component of the package being build. In case of failure, please report by email agata@ipnl.in2p3.fr or in the bug tracker. Remember to give as much as possible all details about the ROOT and the GammaWare version you are trying to compile, the operating system (Linux, Unix, macosx) and the version of the compiler. Once it is finished, you can install it:

```
make install
```

Once it is done, you are ready to work with the GammaWare package. If the destination directories don't exist, they will be created when *make install* is called. More informations about what is built and where it is installed in the INSTALL and README files coming with GammaWare package.

The GammaWare package does not create any binaries and mainly libraries that can be loaded in a ROOT session. To help using the GammaWare, you should configure your environment. For that, you should run the command :

```
root -b -q 'scripts/GwEnv.C()
```

It creates scripts (located in this example in the */tmp* directory) which could be used to configure you environment for GW. Run this script with

```
source /tmp/GwEnv.sh  
or  
source /tmp/GwEnv.csh
```

depending of the shell you are using. In particular, the LD_LIBRARY_PATH should contain the directory where are installed the GammaWare libraries. Check it with:

```
echo $ LD_LIBRARY_PATH
```

It should contains the path to *libraries* given by the command *make showconf*. Keep the GwEnv script and run it whenever you need to work with the GammaWare.

To check that everything is properly installed, go to the *demos* directory and run the following commands:

```
root  
root[0] .x RunTest.C
```

if one of the test fails, please check carefully that all the procedures have been followed correctly

otherwise report.

3.2 Other packages

Narval

To be written

PSA

To be written

PRISMA

In this package are the methods to extract from prisma raw data the physical quantities such as recoil velocity, identification of the nucleus. The package could be found here :

Enter as guest and go to the Gamma/Prisma directory. There is documentation inside the package to configure, compile and use it.

Tracking

To be written

DATA

Some data of test experiments are available here:

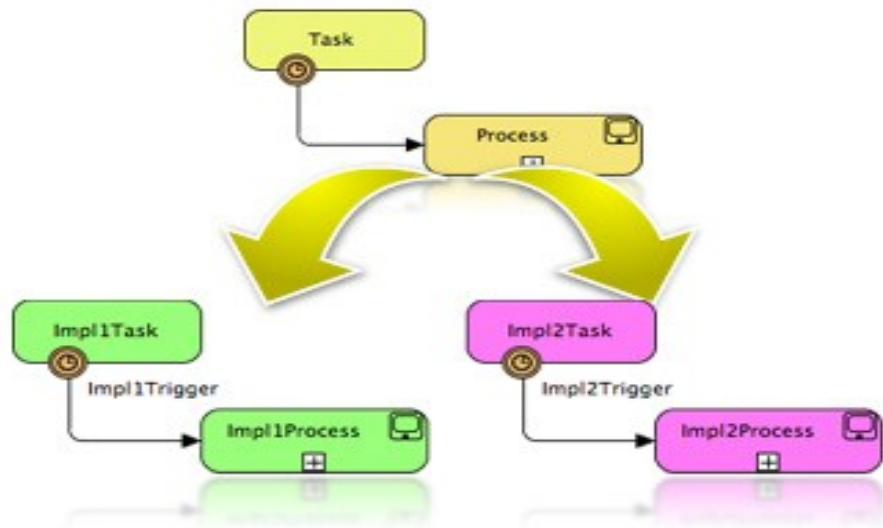
<ftp://t2-srm-03.lnl.infn.it:2121/data>

Even if you can log in as a guest, you should be authorised to access these data. Ask one the local people in Legnaro to get this permission.

ELOG

Elog for test experiments <http://agata-0.lnl.infn.it:8989/>

Chapter 4 - The GammaWare in details



Chapter 4 - The GammaWare in details.....	22
4.1 Overview.....	23
4.2 The Core component	24
4.2.1 The Log System.....	24
4.3 The Physics component.....	24
4.3.1 Links, Level and Level Scheme.....	24
4.3.2 Level Scheme Player.....	25
4.3.3 Correlated Spaces.....	25
4.3.4 Spectrum Player.....	25
4.4 The GEM component.....	26
4.5 The Tools component.....	26
4.5.1 Conversion of histograms.....	26
4.5.2 Event Display.....	26
4.6 The ADF component	27
4.6.1 Introduction.....	27
4.6.2 Keys and Frames.....	29
4.6.3 Interface to Narval.....	37
4.6.4 Interaction of the algorithms with the Data Flow	39
4.7 The ADFE component.....	43
4.7.1 Watchers and FrameDispatcher.....	43

4.1 Overview

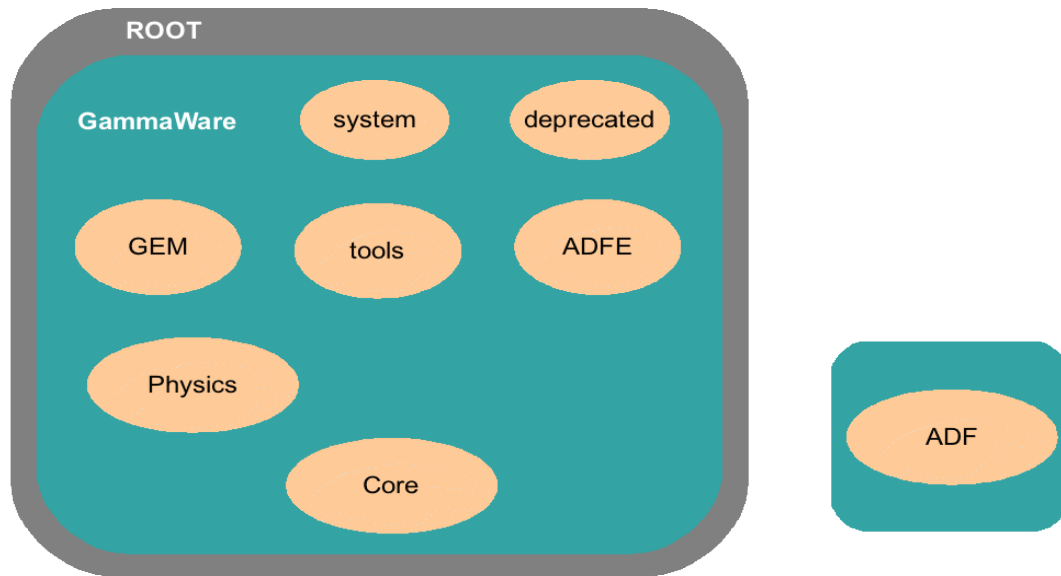


Figure 10: ROOT and the different components of the GammaWare

The GammaWare project is organised in several sub-projects each one having a general purpose. Each sub-project regroups a set of specified C++ classes compiled into a shared library. The package is organized to avoid circular dependencies (see figure 10). Fig gives the different libraries and illustrate more or less the dependencies. The main purpose of the different components is:

- The core sub-project (GWCORE library) regroups all the classes common to all the others components. It defines for instance a log system on which relies the full package.
- The Physics sub-project (GWPHYSICS library) defines the objects manipulated in γ -ray spectroscopy like for instance a Level, a Link (it binds two Levels together) a level scheme.
- The ADF library (GWADF) contains all the facilities to deal with the Agata Data Flow (i.e. mostly reading/writing real data). Because this library is designed to be used online but also possibly in any analysis framework, it does not depend on any other external library, and relies only on the C++ compiler.
- The GEM (Gamma-ray Events Monte Carlo) library (GWGEM) is used to generate events for simulations. It is thus possible to randomly generate discret gamma-ray cascades from a LevelScheme. Facilities to produce files suitable for the Geant4 AGATA code are there also as well as interfaces to other Monte-Carlo (PACE, Evapor).
- The Tools library (GWTOOLS) adds specific tools required for gamma-ray spectroscopy like interactive players for gamma-gamma matrices, facilities to convert 1D, 2D histograms from one framework to another one ...
- The ADFE library (GWADFE) contains all the facilities to deal with the different Data Flow. It extends the ADF library taking advantages on ROOT and the GammaWare.
- The Deprecated library (GWDEPRECATE) contains classes that should not be used anymore but keep it there to help the migration.
- The System library (GWSYSTEM) should give interface to the system if ROOT has not yet provided it.

All the libraries depend on ROOT except the ADF one. This library should stay a pure C/C++ library because it should be plugged also in the AGATA DAQ system (Narval) and in fact in any analysis framework.

4.2 The Core component

In the core component are regrouped all the classes common to all the others components of the whole package.

4.2.1 The Log System

4.3 The Physics component

The package is mainly dedicated to build a level scheme from a correlated space (e.g.: matrix, cube) including the procedures of filtering and of fitting for a given dimension.

4.3.1 Links, Level and Level Scheme

Basically a level scheme contains a number of levels connecting or not by links. The base classes are purely graphical [Gw::Level](#) and [Gw::Link](#). For each link, there is a initial level and a final level associated. A cascade or a band ([Gw::Cascade](#)) is defined as a collection of links and with a given name. Levels and links could be set movable/visible or not. Colors could be set to disentangle between different status of the objects (e.g.: parity, known states, graphically selected, etc...).

4.3.1.1 Links

A link is a graphical object symbolized by an arrow ([TPolyLine](#)) with a given color depending of the know state or its selection. Attached to the level three different labels could give additional informations. These labels could be customized by the user (position and visibility).

```
class Link
{
protected:
...
    TString fVisLabel; // To know if a label should be drawn ok not
    TString fPosLabel; // To know the position of label 0:left; 1:center; 2:right

    TLatex fLabel0;    // Some labels associated with this link
    TLatex fLabel1;    // Some labels associated with this link
    TLatex fLabel2;    // Some labels associated with this link

    Level *fInitial;    // initial level
    Level *fFinal;      // final level

protected:
    Measure<Float_t> fStrength; // strength that binds the two levels
    Measure<Float_t> fTau;      // characteristic time to ge from one level to the other one
...
public:
    virtual void SetVisLabel(const char *);
    virtual void SetLabels(const char *l0, const char *l1, const char *l2);
    virtual Measure<Float_t> & GetTau() { return fTau; }
    virtual Measure<Float_t> & GetStrength() { return fStrength; }

    virtual Level* SetFL(Level *final) { Level *tmp = fFinal; fFinal = final; return tmp; }
    virtual Level* SetIL(Level *initial) { Level *tmp = fInitial; fInitial = initial; return tmp; }
    virtual Level* GetIL() { return fInitial; }
    virtual Level* GetFL() { return fFinal; }
...
};
```

Several classes inherit from the base class **Gw::Link**, namely **Gw::GammaLink** and **Gw::XGammaLink**. The first class contains the information about the nature of the transition, the multipolarity, the energy, the mixing ratio and the conversion factor. The two first observables are object of template class **Gw::Data** with the given information **Gw::InfoData**. The lastest are object of class **Gw::Measure** that hold the value and its associated error (class **Gw::Data**). These observables could be assigned to the labels of the base class. A XGammaLink is a converted gamma-ray link with the list of the associated X-rays.

```
class GammaLink : public Link
{
protected:
...
    Data<Char_t> fEM;          // Electric/Magnetic transition
    Data<UShort_t> fLambda;    // numeric part of the multipolarity

    Measure<Float_t> fEnergy;  // gamma ray energy
    Measure<Float_t> fMixing;  // mixing ratio
    Measure<Float_t> fConv;    // conversion

public:
...
    virtual Measure<Float_t> & GetMixing()      { return fMixing; }
    virtual Measure<Float_t> & GetEnergy()      { return fEnergy; }
    virtual Measure<Float_t> & GetConversion() { return fConv; }

    virtual void SetEM(const char *em= "E2");
    virtual bool IsE() const { return fEM.Get() == 'E'; }
    virtual bool IsM() const { return fEM.Get() == 'M'; }
    virtual UShort_t GetLambda() const { return fLambda.Get(); }

...
};
```

4.3.1.2 Levels

A level is a graphical object (**TLine**) with a given style and color depending of its nature or its parity. Attached to the level four different labels could give additional informations. As for links, these labels could be customized by the user (position and visibility).

```
class Level
{
private:
    TString fVisLabel; // To know if a label should be drawn ok not
    TString fPosLabel; // To know the position of label 0:ul; 1:ur; 2:lr; 3:ll

    TLatex fLabel0; // Some labels associated with this level
    TLatex fLabel1; // Some labels associated with this level
    TLatex fLabel2; // Some labels associated with this level
    TLatex fLabel3; // Some labels associated with this level

    TLine fExtra; // Extra line for links far away from this level
    Bool_t fHasExtra; // flag to know whether the level has an extra line

...
public:
    virtual void SetVisLabel(const char *);
    virtual void SetLabels(const char *l0, const char *l1, const char *l2, const char *l3)
    virtual TLine& GetExtraLine() {return fExtra;}

...
};
```

The class **Gw::NuclearLevel** inherits from the base class (**Gw::Level**) and is characterized by an energy, a live-time, a spin and a parity. The two first observables are objects of class **Gw::Measure**. Spin (**Gw::Spin**) and parity (**Gw::Parity**) are specific objects with a given quantum number (**Gw::QNumber**) and given informations. These observables could be assigned to the labels of the base class. For graphical purposes extra lines could be attached to a level.

```
class NuclearLevel : public Level
{
protected:
...
    Measure<Float_t> fEnergy; // energy and its error
    Measure<Float_t> fT;      // half-life of this level
    Parity fParity;          // parity of this level
    Spin fSpin;              // spin of this level
public:
...
    Spin& GetSpin() { return fSpin ; }
    Parity& GetParity() { return fParity; }

    Measure<Float_t> & GetE()      { return fEnergy; }
    Measure<Float_t> & GetEnergy() { return fEnergy; }
    Measure<Float_t> & GetT()      { return fT; }
...
};
```

4.3.1.3 Level Scheme

A level scheme is a set of collections (**TList**) of levels, links and cascades. These objects can be added via the Add methods. Whenever one of the object is selected, one can get the current object via the given methods. The level scheme has a name. Some additional objects could be added as a reference, a list of text or a list of arrows.

One can import already existing level scheme (format AGS and ENSDF supported). The reading of the file is done with the help of the **Gw::RadLevelSchemeReader** and **Gw::EnsdfLevelSchemeReader** classes respectively. The interactivity with the canvas is done via a level scheme player.

```
class LevelScheme : public TObject
{
...
private:
    Link* fCLink;      /// pointer to current link
    Level* fCLevel;    /// pointer to current level
    Cascade* fCCascade; /// pointer to current cascade
...
protected:
    TList fLevels;      // list of levels
    TList fLinks;       // list of links
    TList fCascades;    // list of cascades

    TLatex fName;       // LevelScheme's name
    TLatex fReference;  // some information concerning the origin of this level scheme
    TList fTexts;       // a list of additional TLatex text to add some comments
    TList fArrows;      // a list of additional TArrow to add some arrows on the level scheme

    BaseLSPlayer* fPlayer; /// level scheme player pointer
public:
    virtual Int_t Import(const Char_t *, Option_t *);
    virtual void Draw(Option_t *opt = "");
    virtual Int_t DistancetoPrimitive(Int_t px, Int_t py);
    virtual void ExecuteEvent(Int_t event, Int_t px, Int_t py);
...
};
```


4.3.2 Level Scheme Player

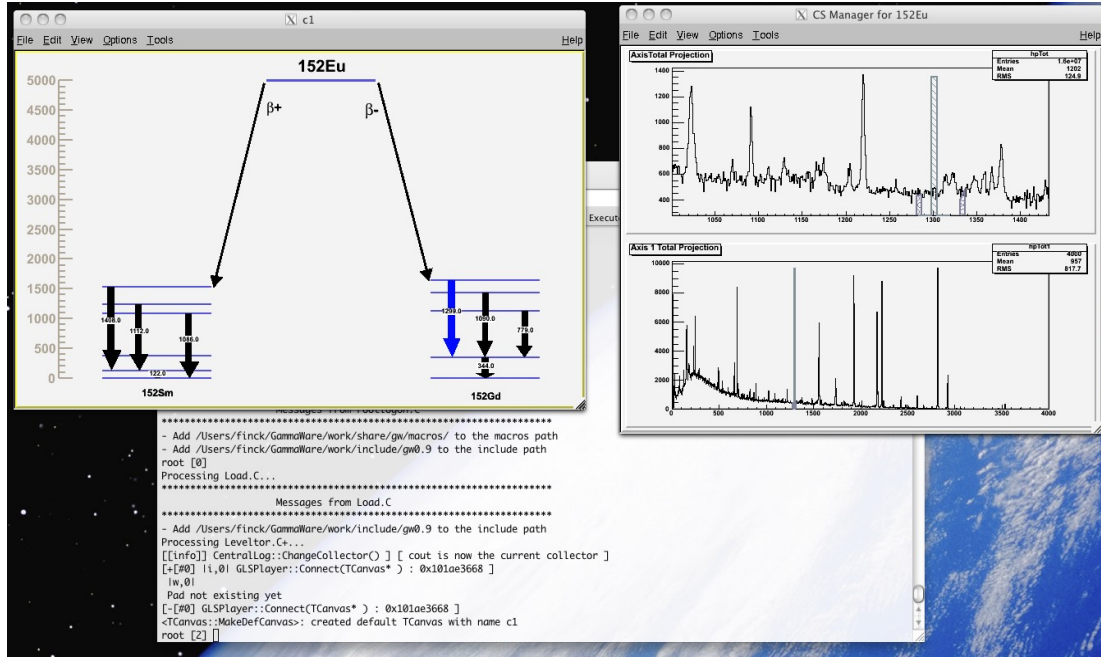


Figure 11: Example of a level scheme built with the player with two associated correlated spaces. On the left the level scheme with a selected link, on the right side the two total projections of the correlated space with the gate associated to the selected link.

This package regroups tools for creating/managing level schemes. The base class **Gw::BaseLSPlayer** handle mainly the interaction with the mouse/keyboard and the interface with the level scheme class. The main class is **Gw::GLSPlayer** (Graphical Level Scheme Player). Using the context menu of ROOT, object parameters could be set or modified via those menu simply by activated the popup menu via the mouse click. Passing over objects (namely link or level) a tooltip shows up given primary informations about the object. A specific level scheme player menu could be activated by clicking onto the energy axis of the scheme. One can make the menu active over the whole canvas by pressing key 't'. This menu is subdivided in specific sub-menus for each type of actions on cascades, level, link and on level scheme. One can select/deselect a object by pressing key 's', the whole level scheme could also be selected/deselected by pressing key 'a' (see figure 11).

Recently one can register a correlated space object (see next section) and apply gates from selected links. A histogram with the gated projection is drawn. A sub-menu for Correlated Space (CS) appears in the GLSPlayer menu. One can register a correlated space stored in a root file and apply gate with simply click 'p' key.

The GLSPlayer could also provides tools for plotting moment of inertia when selecting a given list of links or plotting gates onto a existing spectrum. A video ([todo](#)) shows to use the different functionality of the player.

4.3.3 Correlated Spaces

These spaces contain correlated informations for a given dimension. The main features are filling and projection using a filter (e.g.: gating). The associated total projections and filtered projections for each dimension are also member of the base class **Gw::CorrelatedSpace**. A distinguish for symmetrical and asymmetrical spaces is made with the help of the **Gw::SymCorrelateSpace** and **Gw::ASymCorrelateSpace** classes.

```

class CorrelatedSpace : public TNamed
{
protected:
    Int_t fDimension;          // dimension of the correlated space
    TList* fHistoTotList;      // list of histograms for total projection
    TList* fHistoList;         // list of histograms for projection
    Int_t fStatus;             // status of the tree
...
public:
    virtual void Fill(const Double_t* /*Xn*/, const Double_t /*weight*/ = 1) = 0;
    virtual void Project(CSFilter* /*filter*/, Option_t* /*axis*/ = 0) = 0;
    virtual void Project(Option_t* /*axis*/, Option_t* /*gateName*/ = "Gate0") = 0;

    virtual void SetDimension(Int_t dimension) = 0;
    virtual Int_t GetDimension() const = 0;

    virtual void SetBins(Int_t bin, Double_t min, Double_t max, Int_t axis = 0) = 0;

    virtual TH1D* GetHisto(Option_t* axis = "x") = 0;
    virtual TH1D* GetHistoTot(Option_t* axis = "x") = 0;
...
};

```

A specific two dimensional correlated space using a **TTree** root could be used via the **Gw::CorrelatedSpaceTree** class. This class inherits from the base class **Gw::SymCorrelatedSpace**.

```

class CorrelatedSpaceTree2 : public SymCorrelatedSpace
{
protected:
    TTree* fTree;              // tree that contains the matrix
    TString fBranchName;       // name of branch where the matrix is stored
...
public:
...
    virtual TBranch* SetAddress() = 0;
    virtual void SetBranch() = 0;
    void SetBranchName(TString name = "mat1") { fBranchName = name; }
    const Char_t* GetBranchName() { return fBranchName.Data(); }

    void Fill(const Double_t* Xn, const Double_t weight = 1.);
    void FillRandom(Int_t dim);
    void FillFromH2(TH2* h2);

    void Project(CSFilter* filter, Option_t* axis = "X");
    void Project(Option_t* axis = "X", Option_t* gateName = "Gate0");
...
};

```

Since the use of tree creates a dependency of the type of the object stored, dedicated classes has been implemented for each current type, namely for float, double and integer, respectively with **Gw::CorrelatedSpaceTree2F**, **Gw::CorrelatedSpaceTree2D** and **Gw::CorrelatedSpaceTree2I**. The tree could be filled from a **TH2** histogram, from an array or being filled randomly. This correlated space is suited for big space or while using more than one space since it do not load the full space in memory. For small space (typically a 4k matrix of float) better use the correlated space using a **TH2** root object (**Gw::CorrelatedSpaceTH2**). The type of the data is given as argument in the constructor. This class inherits from the base class **Gw::ASymCorrelatedSpace**. In the base class **Gw::CorrelatedSpace** an import method allows to convert from one type of correlated space

to another.

The projection of the axis is done with the help of the base class **Gw::CSFilter**, that holds the array of the filter and filter condition. The implementation for cubic gating is performed with the **Gw::CubicFilter** class. A correlated space manager class has been added (**Gw::CSManager**), it allows to make projections from different correlated spaces with a given filter. It also take care about the drawing of the different spectra onto a specific canvas. Collecting and drawing of the gates are performed with this manager.

```
class CSFilter : public TNamed
{
protected:
    Short_t fCondition;          //!< number of conditions for gating
    Short_t fDimension;          //!< dimension for projection
    TArrayD fGateInf;            //!< inferior gate array
    TArrayD fGateSup;            //!< superior gate array
...
public:
...
    virtual Int_t IsInside(Double_t* /*energies*/, Int_t /*mult*/, Short_t* /*inGate*/) { return 0; }

    Int_t GetNofGates() const { return fGateSup.GetSize(); }
    Double_t GetGateInf(Int_t idx) { return fGateInf.At(idx); }
    Double_t GetGateSup(Int_t idx) { return fGateSup.At(idx); }

    void AddGate(Double_t gateInf, Double_t gateSup, Option_t* opt = "update");
    void ImportGates(const Char_t* name, Option_t* opt = "update");

    Short_t GetCondition() const { return fCondition; }
    void SetCondition(Short_t cond) { fCondition = cond; }

    Short_t GetDimension() const { return fDimension; }
    void SetDimension(Short_t dim) { fDimension = dim; }

...
};
```

4.3.4 Spectrum Player

This player enables fitting procedure adding background with pre-defined or user defined functions onto an existing spectrum with a given dimension (for the moment only one dimensional spectra are implemented). The base class **Gw::BasePeak** holds the basic informations about a peak, regardless of its dimension, namely intensity, height and dimension. All methods needed for the fitting procedure are already declared but in purely virtual way.

```
class BasePeak : public TNamed {
public:
...
    virtual void SetFunction(const char* nameFunc = "gaus") = 0;
    virtual void SetUserFunction(const TF1* func) = 0;
    virtual void SetUserBkgFunction(const TF1* func) = 0;

    virtual void Fit(Option_t* optFit = "RN", Option_t* optBkg = "lin") = 0;
    virtual void FitCombined(TList* peakList, Option_t* optFit = "RN", Option_t* optBkg = "lin") = 0;

    virtual Double_t GetPosition(Option_t* axis = "X") const = 0;
    virtual Double_t GetFWHM(Option_t* axis = "X") const = 0;

    virtual void SetPosition(const Double_t position, Option_t* axis = "X") = 0;
    virtual void SetFWHM(const Double_t FWHM, Option_t* axis = "X") = 0;
...
};
```

```
protected:
    Double_t fHeight;      ///< height of the Peak
    Double_t fIntensity;    ///< intensity of the peak
    UShort_t fDimension;    ///< dimension of the peak
...
};
```

The implementation for one dimensional peak is done in the class **Gw::Peak1D**. Such a peak is graphically defined with a **TPolyLine** symbolized by a customized triangle shape (see figure 12).

```
class Peak1D : public BasePeak {
...
public:
...
    void SetHeight(const Double_t height);      // *MENU*
    void SetIntensity(const Double_t intensity); // *MENU*
    void SetPosition(const Double_t position, Option_t* axis = "X"); // *MENU*
    void SetFWHM(const Double_t FWHM, Option_t* axis = "X"); // *MENU*

    void SetBackground(Double_t bgLeft1, Double_t bgLeft2, Double_t bgRight1, Double_t bgRight2,
        Double_t bgLevelLeft1, Double_t bgLevelLeft2, Double_t bgLevelRight1, Double_t bgLevelRight2);

    virtual Double_t GetPosition(Option_t* axis = "X") const;
    virtual Double_t GetFWHM(Option_t* axis = "X") const;

    virtual void SetFunction(const char* nameFunc = "gaus");
    virtual void SetUserFunction(const TF1* func);
    virtual void SetUserBkgFunction(const TF1* func);

    virtual void Fit(Option_t* optFit = "RN", Option_t* optBkg = "lin");
    virtual void FitCombined(TList* peakList, Option_t* optFit = "RN", Option_t* optBkg = "lin");
...
private:
    Double_t fPosition;      // Position of the Peak in energy
    Double_t fFWHM;          // FWHM of the Peak
    Double_t fBkgLeft1;      // lower limit for left side background
    Double_t fBkgLeft2;      // upper limit for left side background
    Double_t fBkgRight1;     // lower limit for right side background
    Double_t fBkgRight2;     // upper limit for right side background

    TPolyLine* fPolyLine;    // pointer to polyline
    TF1* fSigFunc;           // pointer to signal function
    TF1* fBkgFunc;           // pointer to bkg function
    TF1* fPeakFunc;          // pointer to signal+ bkg function
...
};
```

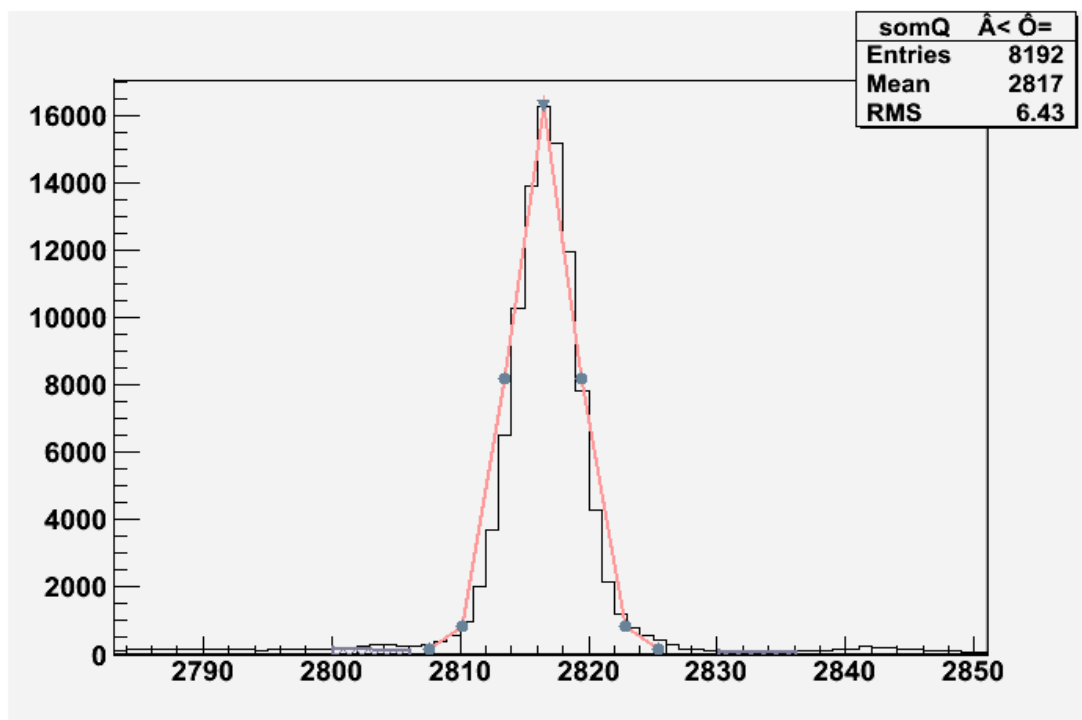


Figure 12: Displayed one dimensional peak with background onto a spectrum

One can set graphically the height (triangle), the width and the base (bullet) of the peak. The background line could be set moving the corresponding corners. The goal is to set initial parameters for the fitting procedure. The user has to set the signal function (default wise set to a gaussian shape) and the background function (default wise set to a first order polynomial). The fit is performed and the result is stored in the associated members of the class and the peak is redrawn with these new parameters. The color of the polyline, the markers and the fitting function could be change via static methods.

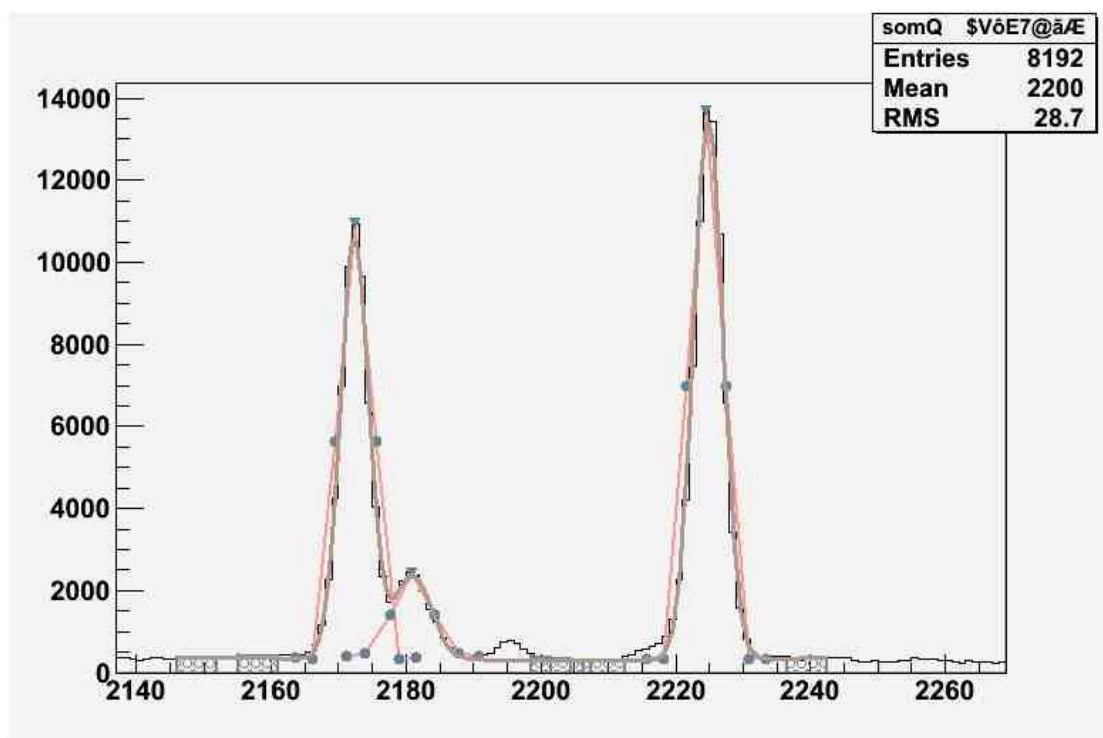


Figure 13: A multiple peaks fit display.

A collection of peaks could be fitted with the help of the spectrum player classes. The base class **Gw::BaseSpectrumPlayer** connects to the given canvas, manages the list of peaks and contains the base methods. A dedicated class **Gw::PeakCreator** manages the mouse and keyboard actions. Pressing key 'a' draw a peak a mouse position, pressing key 'A' draw a peak a mouse position with background and pressing key 'p' put a polyline point at mouse position. The fit of one dimensional peaks is performed once peaks are set, the class can fit all defined peak taking into account the narrowing of the peaks (see figure 13). One can popup fit panel menu pressing key 'f'.

There are two classes inherits from this base class. The class **Gw::GSPlayer** has a graphical method to put peak at mouse position in a sliding way is implemented. A line scroll over a given range of the histogram, pressing key 's' put a peak at line position. To continue press any other key. Background points could be set pressing key 'p'.

A second level scheme player (**Gw::RootSpectrumPlayer**) exists based on **TSpectrum**, it finds the peak respect to given parameters (threshold, intensity and width). The background subtraction also based on this root class.

The collected peaks can then be fitted.

4.4 The GEM component

4.5 The Tools component

This package provides mainly tools for manipulating spectra (converter, calibrating) but also an event display.

4.5.1 Conversion of histograms

This sub-package allows to read back given histograms and/or to convert them in another type. Supported type are presently are radware, gpsi, euroball and ascii format. The main class that handles the list of a given type of histograms is **Gw::HistoDB**. A histogram can simply be extracted from list (e.g.: a list of spectra in a directory) by its name. Reading from database or writing to database is done respectively with the shift operators '>>' and '<<'.

The effective reading and writing is done with the base class **Gw::HistoConverter**. The instantiation of the different types of histograms is done during the opening of the database. The classes are respectively **Gw::RadConverter**, **Gw::EGConverter**, **Gw::GPSIConverter**, **GW::ASCConverter** for radware, euroball, GPSI and ascii format. For this latter the format has to be specified (see header file for details) with the SetOption method.

An example how to use these classes could be find in section demos/tools.

4.5.2 Event Display

The GW display is base on the **TEve** ROOT classes using the OpenGL graphics libraries. Basically an event display is a set of informations plotted in a given geometry for each event (see figure 14). Informations are stored in a Tree (base class **Gw::BaseEventContainer**). The class **Gw::AgataEventContainer**, defines the kind of informations, namely hits and tracks for specific type of particle/gamma. Each type has its own branch in the Tree, customized branches could be added. The base class (**Gw::BaseEventDisplay**) of the display inherits from the class **TEveEventManager**, it manages the browsing through the event and the loading of the geometry parts. The main class **Gw::AgataEventDisplay**, contains the list of hits (**Gw::AgataHitDisplay**) and the list of tracks (**Gw::AgataTrackDisplay**). Those classes are built from the classes

respectively from **TEveQuadSet** and **TEveBoxSet**.

```
class AgataEventDisplay : public BaseEventDisplay
{
private:
    static AgataEventDisplay* fgInstance; // static instance of class

public:
    ...
    static AgataEventDisplay* Instance();

    void FirstEvent(); // *MENU*
    void NextEvent();  // *MENU*
    void PrevEvent();  // *MENU*

    void ShowDisplay();

    AgataEventContainer* GetEventContainer() const { return fAgataContainer; }

    const Char_t* RegisterBranch(Option_t* type = "");
    ...
private:
    AgataEventContainer* fAgataContainer; // container of hits/tracks

    TList* fQuadHitList;           // list of quad to display hits
    TList* fLineTrackList;         // list of line to display tracks

    ...
};
```

The Eve display is customized with a GUI info panel, it allows to navigate through the event with a simple click on the button reset, backward and forward. With the double selection of Eve (alt or ctrl click) one can have the informations concerning a selected hit/track (see figure 14). An important point is that the Agata Event Display class is defined as a singleton, only one display could be opened, respect to the fact that the **TEveManger** pointer is also a singleton.

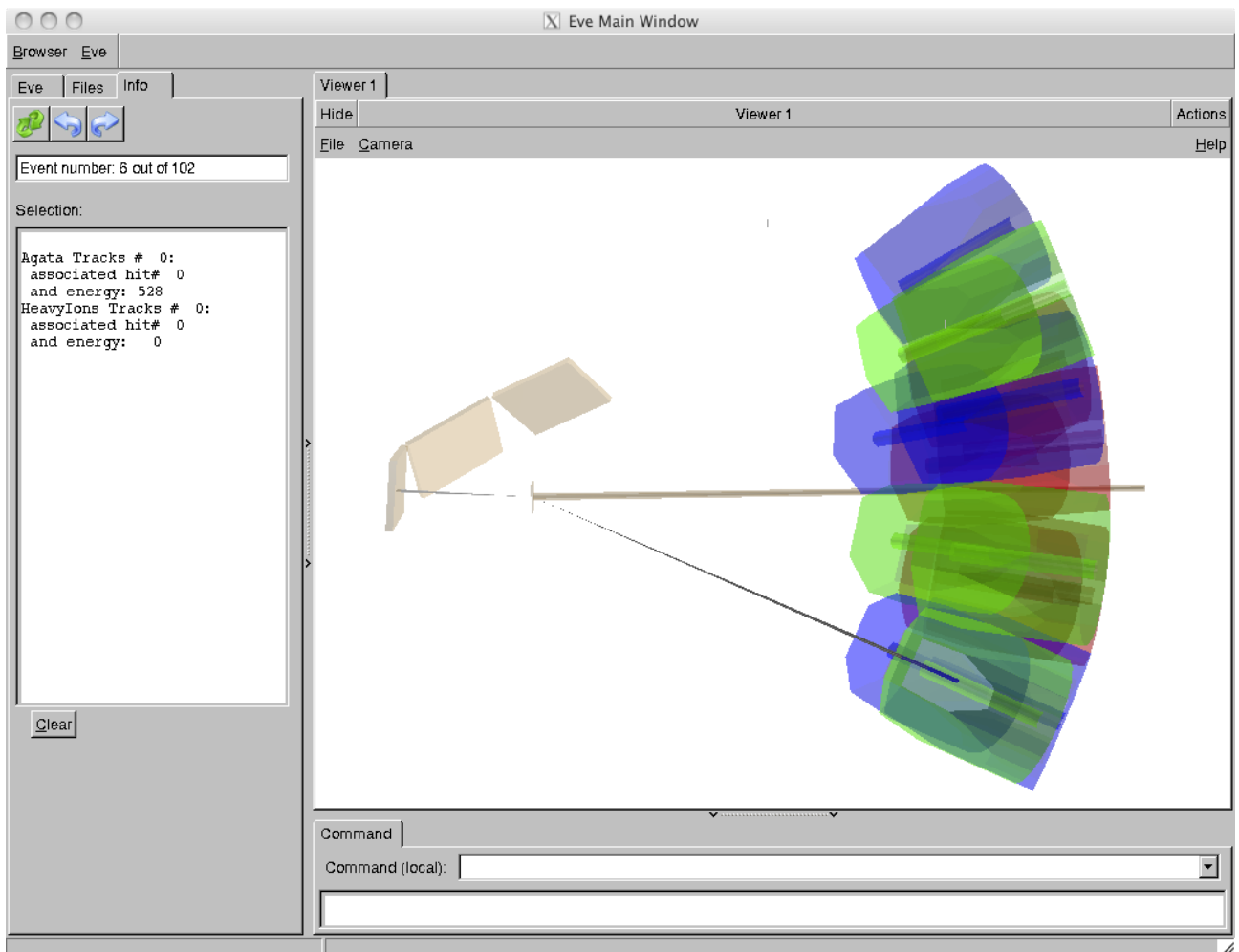


Figure 14: Example of an event display with Agata and Dante detector. A track is observed in each detector. Also depicted the beam pipe and the target.

4.6 The ADF component

4.6.1 Introduction

The AGATA Data Flow is highly processed by different algorithms, the most important ones being the PSA and the Tracking algorithms. To process efficiently online such a data flow, the AGATA DAQ box relies on Narval (ref): it implements a fully distributed system thanks to the ADA language. In Narval terminology, an algorithm processing the data flow is called an **Actor**. Because Narval is a distributed system, the different **Actors** can run in different computers. The data are sent from one **Actor** to another one through the network using large buffers.

There are different types of **Actors** (see Figure 11): a **Producer** produces data, a **Consumer** consumes them while a **Filter** consumes data in order to produce new data. **Actors** could be chained each others in different topologies managed by Narval through configuration scripts. Two examples are given in Figure 12. The first one represents a simple topology in which a single **Producer** (data coming out from an AGATA crystal) sends data to a **Filter** (PSA) that processes them before sending them to a **Consumer** (dumps data on disk). In the second topology, data coming out from two crystals are processed by a **Filter** (PSA), merged, then the output is processed by another **Filter** (Tracking) itself sending the tracked gamma-rays to a **Consumer** (dumps data on disk).

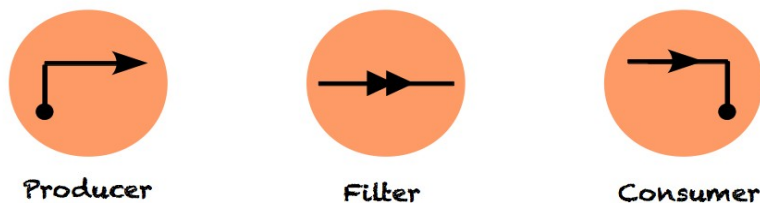


Figure 15: Different kind of Actors processing the AGATA data flow

Narval is written using the ADA language. It would not be convenient to have to develop all the **Actors** processing the AGATA Data Flow in such a language. Fortunately, ADA provides facilities to link libraries developed using other languages. Thus, it is possible to define a C/C++ **Actor** and load it in Narval. For that, a precise interface (described in the following document²) should be provided in the loaded libraries.

Depending of its type, a particular **Actor** receives an input and/or an output (large) buffer(s) in which it should:

- extract from the input block the particular set of data it needs to work on
- add to the output block the data it has produced
- deal with data that are not required (for instance, move them from input to output)

Any data block is composed of sequences of **Frames**. A **Frame** is a buffer which starts with a **Key** followed by the **Frame** content produce/consume by a given algorithm. The **Key** part gives global informations about the content. One of the most obvious type of information is the length of the **Frame**. Thus any **Frame** starts with its full length. In the input block, a **Filter** should then identify a particular **Frame**, read its content, process it, produce a new **Frame** and add it to the output block.

2 - Agata PSA and Tracking Algorithm Integration, J. Cresswell and X. Grave, 2nd Draft

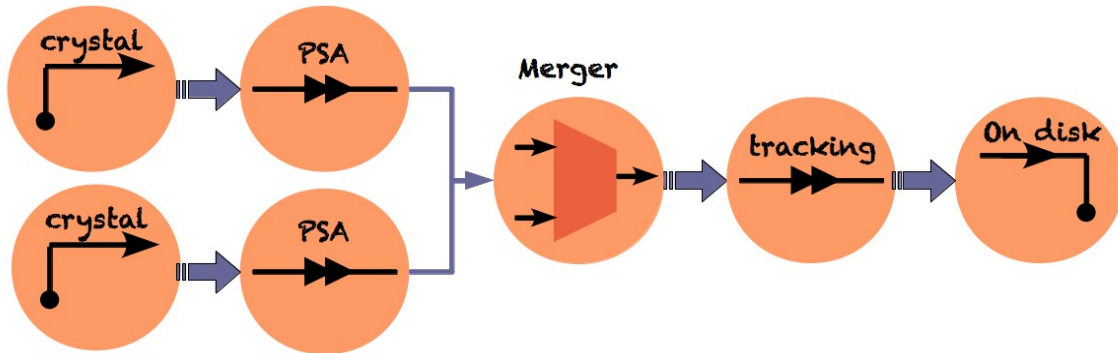
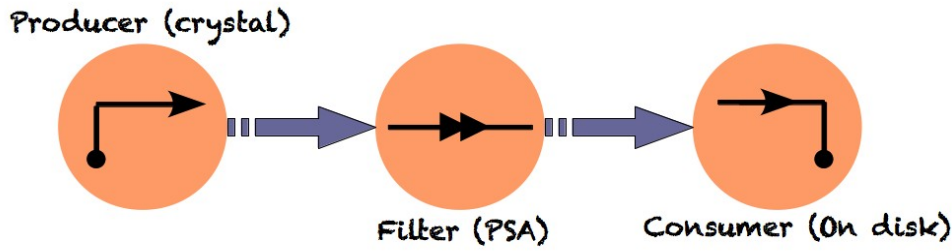


Figure 16: Two different topologies

To help an algorithm to interact with the AGATA Data Flow, a C/C++ library, called ADF, has been developed (see the design proposal³). This library is part of the GammaWare but could also be used in a standalone mode (see Chapter 2). This is mandatory since ADF is loaded online for **Actors** processing the data flow.

This library defines many objects to help interacting with the AGATA Data Flow. A virtual interface to Narval is also providing so that the same code could run online but also offline through emulators. It helps for developing/debugging/testing of algorithms since this cannot be done easily in Narval. The main concepts and C++ classes are presented in this section.

4.6.2 Keys and Frames

As pointed out, Narval feeds an algorithm (an **Actor**) with series of large buffers. These buffers are composed of sequences of **Frames**. A **Frame** starts by a **Key** that contents general informations concerning the data part. The **Keys** are here to structure the data flow (mapping/identifications). It helps also to manipulate **Frames** at a 'global' level. One of the most obvious informations is the length of the **Frame**. Thus a **Key** starts with such a field (see Figure 13).

There are different kind of **Keys** and **Frames**. As well **Keys** and **Frames** have an unique version number associated in the form of a pair of integers (Major #, Minor #). This is required since algorithms are likely to change over time and the real data may be a little bit different ... but not completely different. For instance, the PSA algorithm should produce a list of **Hits** but some versions of the algorithm may be able to provide errors on the positions while others not. In the first case, the errors are written in the **Frame** while in the second case they are not. A given **Key** or **Frame** is uniquely defined by two strings and a version number:

- A general category (also name of the Factory that creates them, see section)

³ - A library of services to decode/encode data through the AGATA Data Flow (see DAQ web site)

- A given type in this category
 - A version number
- Ex: “Agata”, “[data:psa](#)”, Version(1,0)

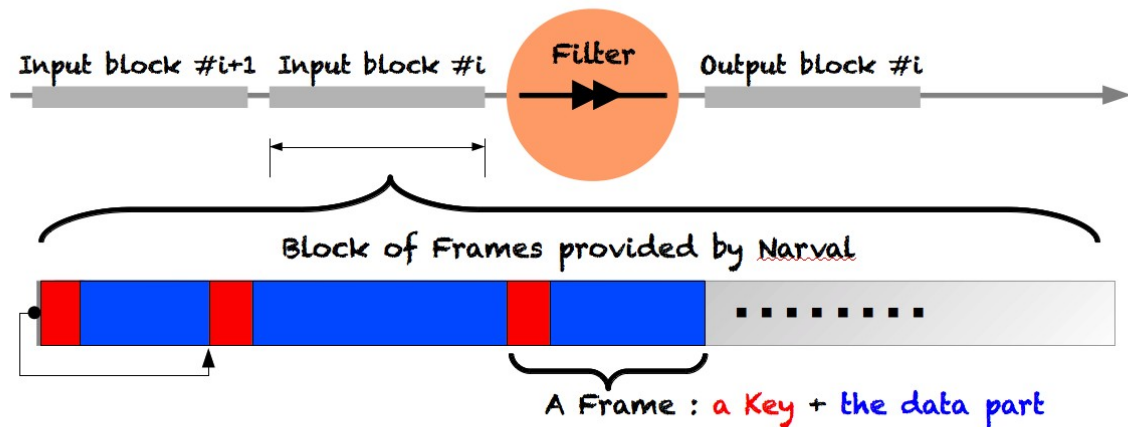


Figure 17: The AGATA Data Flow is composed of Frames

4.6.2.1 Keys

A **Key** contains the basic informations to map/structure the data flow. A C++ base class which provides methods to access this informations is available in ADF. Here are some methods to Set/Get the different lengths.

```
class Key
{
public:
...
    virtual void SetDataLength(UShort_t);
    virtual void SetDataLength(UInt_t);
...
    virtual UInt_t GetDataLength() const ;
    virtual UInt_t GetFrameLength() const ;
...
    virtual UInt_t GetKeyLength() const ;
...
};
```

For the AGATA Data Flow, ADF defines an **AgataKey** (inherits from a **Key**) with different versions which include more and more fields. In the following Table is given the active fields (in blue) and the associated length for each version. The adopted convention (comes from root) is:

- Char_t (Uchar_t) : 1 byte
- Short_t (Ushort_t) : 2 bytes
- Int_t (UInt_t) : 4 bytes
- Long64_t (Ulong64_t) : 8 bytes
- Float_t : 4 bytes
- Double_t : 8 bytes

Key Version	Key content (what is really written in the data flow)				Key length
(0,0)	Frame Length UInt_t	Message type	Event Number	Timestamp	4 bytes
(1,0)	Frame Length UInt_t	Message type UInt_t	Event Number	Timestamp	8 bytes
(2,0)	Frame Length UInt_t	Message type UInt_t	Event Number UInt_t	Timestamp	12 bytes
(4,0)	Frame Length UInt_t	Message type UInt_t	Event Number UInt_t	Timestamp ULong64_t	20 bytes

Tableau 1: The different AgataKey as defined in ADF

The message type (4 bytes) qualifies the content of the **Frame**. A given type of **AgataKey** is specified when created by the factory using a name. The message type is coded with the convention (see ADF.conf file in the distribution):

```
# 0xFAXXYYZZ
# FA means Flow/Frame for AGATA
# XX = Sub-detector number (associated to a given factories)
# 01 for Agata (Agata factory)
# YY = 01 (Frame type)
# 01 data (data produce by online algorithm)
# 11 meta (general data, does not depend on an algorithm)
# 02 conf (configuratio frame)
# 03 info
# ZZ = Producer ID, should be unique in the data flow
```

The **AgataKey** class provides methods to access the different fields :

```
class AgataKey : public Key
{
public:
...
    //! To get the message type encoded
    virtual UInt_t GetMessage() const ;
    //! To get the event number encoded
    virtual UInt_t GetEventNumber() const ;
...
    //! To set the event number
    virtual void SetEventNumber(UInt_t) ;
...
    //! To get/set the timestamp
    virtual ULong64_t GetTimeStamp() const = 0;
    virtual void SetTimeStamp(ULong64_t) = 0;
};
```

Keys are used to map and structure the Data Flow. They are used by the Event Builder and the Merger which don't have to know anything about the data part to work properly. As well, an ancillary **Frame** could be added to the system quite easily just by having a buffer starting by a proper **Key**.

Here is an sample giving the message type associated to some of the currently defined **Frames** on

the Agata Data Flow. The last column corresponds to the mask used to test the message type.

agata	0xFA000000	0xFF000000
conf	0xFA000200	0xFF000F00
conf:global	0xFA000200	0xFFFFFFFF
conf:crystal	0xFA010201	0xFFFFFFFF
conf:psa	0xFA010202	0xFFFFFFFF
conf:ranc0	0xFA0102A0	0xFFFFFFFF
conf:ranc1	0xFA0102A1	0xFFFFFFFF
conf:ranc2	0xFA0102A2	0xFFFFFFFF
data	0xFA000100	0xFF000F00
data:crystal	0xFA010101	0xFFFFFFFF
data:ccrystal	0xFA010111	0xFFFFFFFF
data:psa	0xFA010102	0xFFFFFFFF
event:data:psa	0xFA010103	0xFFFFFFFF
event:data	0xFA010104	0xFFFFFFFF
data:tracked	0xFA010105	0xFFFFFFFF
data:ranc0	0xFA0201A0	0xFFFFFFFF
data:ranc1	0xFA0201A1	0xFFFFFFFF
data:ranc2	0xFA0201A2	0xFFFFFFFF
meta	0xFA001100	0xFF00FF00
meta:vertex	0xFA011100	0xFFFFFFFF

4.6.2.2 Frames

A **Frame** is a coherent unit of data, typically it could be the output produced by an algorithm or an event. There are different kind of **Frames** :

- **DataFrame** (i.e. **PSAFrame**, **TrackedFrame** and **CrystalFrame**). This kind of **Frame** contents the Data consumed/produced by an algorithm. It adds an interface so that the user can easily access the data part without taking care about how it is really encoded.
- **RawFrame**. This kind of **Frame** is just a buffer in which the user could stream whatever he needs. Thus, there is not interface and the user has to encode/decode himself the content.
- **ConfigurationFrame**. This kind of **Frame** contents a sequence of ascii characters (like a string or an ascii file). They are used to exchanged extra-informations between the different algorithms processing the data flow.
- **CompositeFrame**. This kind of **Frame** is composed of other **Frames**.

A C++ base class **Frame** is provided in the ADF library. All other kind inherits from it, implements the methods and add to it specifications.

```
class Frame
{
public:
...
    //! To get the Key associated to this frame
    virtual Key *GetKey();
    //! total length for that frame
    virtual UInt_t GetLength() const ;
...
    //! Reset the current frame
    virtual void Reset() ;
    //! FastReset the current frame, means the data part keep the previous values
    virtual void FastReset() ;
...
    //! It reads the content into dedicated structures from the Frame (data part). It returns the
    number of bytes read (data part)
    virtual UInt_t Read() ;
    //! It writes to the Frame the content of the dedicated structures. It returns the number of
    bytes written (data part)
    virtual UInt_t Write() ;
...
    //! tells if this frame is a composite frame i.e. if it is composed of sub-frames
```



```

virtual Bool_t IsComposite() const ;
//! Scan this Frame. If it is a composite frame, it looks for the keys of sub-frames
virtual UInt_t Scan() ;
//! Returns one of the sub-key in case this is a composite frame
virtual const Key *GetSubKey(UInt_t) const ;
//! Returns the number of sub-frames composing this frame. Scan have to be called first
virtual UInt_t GetNbSubFrame() const
...
};

```

The following paragraphs illustrate the differences between these different types and how to use them.

DataFrame

DataFrame

k	1100010101010101000110010010
---	------------------------------

While several versions of an algorithm (i.e. PSA) are likely to be used in AGATA, the produced data are likely to be almost the same. Think of the PSA algorithm has a **Actor** producing a list of **Hits** from the 36+1 **Signals** corresponding to the 36 segments and the core. The output data are stored in a **DataFrame** (**PSAFrame in this case**) which is a **Frame** but with also convenient methods for the algorithms to Set/Get these data to/from the **Frame**. Of course the **DataInterface** depends on the algorithm producing the data. This is illustrated in the picture .

Once a algorithm has an input **Frame** to be treated, here a **CrystalFrame**, a call of the Read method loads from the underlying buffer the data into high level objects (structures) which are used by the PSA algorithm to produce psa-like data i.e. mainly a list of **Hits**. Once the processing is over, a call of the Write method streams in the output Frame (here a **PSAFrame**) the produced objects.

A given type of algorithm (i.e. Tracking) should produce the same type of data. Thus, in principle, the **DataInterface** is general enough for many different implementations and/or versions of an algorithm. The **Fame** version in this case is related to what is truly written in the **Frame**. For instance, the PSA algorithm is supposed to produce a list of **Hits**. Depending of the implementation, this list could come with, for each interaction point, the errors associated. It would be inefficient to code in buffers the errors if they are not produced by the algorithm ... a lot of 0 for nothing ! One could then decide to have given specific versions numbers for **Frames** with errors⁴. As well, the Tracking algorithms may deal or mot with errors on positions. In this case, the errors are not written (thus read) in (from) the Frame. However, the interface to a Hit object is general and at any time one can access to errors using the provided methods. In case the errors are going on the Data Flow an algorithm could get them. Otherwise it will have the default value, likely to be always 0. The way an **Actor** deals with **Frame** version numbers is explained in section 4.6.4.

⁴ The policy concerning versioning has not yet been fixed

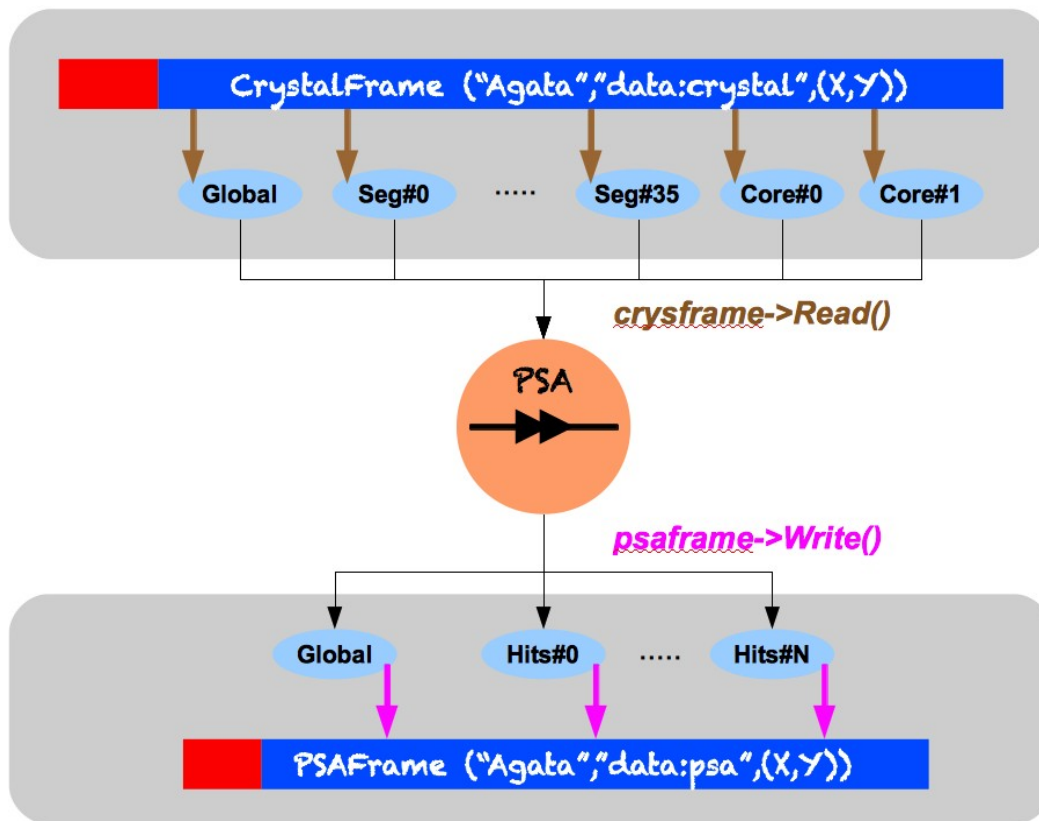


Figure 18: The DataInterface for a DataFrame

All **DataInterface** inherits from a base class **DataInterface**. It provides an entry point to a **Global** object which could be seen simply as a list of named item or data. Indeed it is convenient to give the possibility for any algorithm to set/get to/from the data flow one of several variables tagged by a name. For instance, the algorithm producing the **CrystalFrame** could add an integer **CrystalID** to identify the position of the crystal in the full AGATA array. As well the Tracking algorithm needs this information to transform the positions from a local reference system to the global one. All algorithms are also likely to produce a **Status** flag to keep track of the quality of the treatment. The way to play with the **Global** object is explained in the next sections. Here are some methods associated to **DataInterface** and **Global**.

```
class DataInterface
{
public:
...
    virtual GObject *Global();
...
};

class GObject
{
public:
...
    virtual Bool_t AddItem(ANamedItem *item);
    template <typename Data_t> const NamedItem<Data_t> *Get(const char *name);
    template <typename Data_t> Bool_t Link(const char *name, Data_t *external_address);
...
};
```

ADF defines three different **DataInterface** for the three different kind of Frames managed by the

AGATA Data Flow: **CrystalInterface**, **PSAInterface** and **GammaTrackedInterface**. Here is a sample of the methods available for the different data interfaces.

```
class CrystalInterface : public DataInterface
{
public:
...
    virtual GeSegment *GetSegment(UShort_t) ;
    virtual GeCore *GetCore(UShort_t) ;
...
};
class PSAInterface : public DataInterface
{
public:
...
    virtual Hit *NewHit() ;
    virtual UShort_t GetNbHits() const ;
    virtual Hit *GetHit(UShort_t) ;
    virtual const Hit *GetHit(UShort_t) const;
...
};
class GammaTrackedInterface : public DataInterface
{
public:
...
    virtual Bool_t SetNbGamma(UShort_t how_many, Bool_t do_reset = true) ;
    virtual TrackedHit *NewGamma();
    virtual UShort_t GetNbGamma() const;
    virtual TrackedHit *GetGamma(UShort_t) const;
...
};
```

As it can be seen, the **CrystalInterface** gives access to the different Signals (**GeSegment** and **GeCore**), the **PSAInterface** manages a list of Hits while the **GammaTrackedInterface** allows to deal with **trackedHit** i.e. a **Hit** corresponding to the reconstructed gamma-ray from a trace of **Hits**.

RawFrame

DataFrame

k	1100010101010101000110010010
---	------------------------------

ADF provides an interface to have access to the data contained in a **Frame**. It may not always be the case. For instance an ancillary is likely to produce data with its specific format not known by ADF. A **RawFrame** provides the minimal interface i.e. just a access to the underlying buffer so that the user can stream whatever he needs. Once the buffer has been filled, the **Write** method just compute the right length of the **Frame**.

ConfigurationFrame

ConfigurationFrame

k	Some informations in ascii format
---	-----------------------------------

A **ConfigurationFrame** is just a **Frame** composed of an ascii string (it could be also the contain of a ascii file). It is a way to exchange between different **Actors** human readable kind of data. Thus the interface gives access to the string. Once the string has been filled, the **Write** method should be called to write the in-memory string into the **Frame**. As well, the **Read** method loads from the **Frame** the corresponding string.

```
class ConfigurationFrame : public ConcreteFrame
{
public:
...
    std::string &String()
```

```

    { return fString; }
...
};

```

CompositeFrame



A **CompositeFrame** is a **Frame** composed of **Frames**. This is required to build for instance events. The PSA algorithm transforms the **Signals** (CrystalFrame) coming out from any germanium crystals to a list of **Hits**. The goal of the event builder is to associate in a single **Frame** (a **CompositeFrame**) all the **PSAFrame** based on the event number or the timestamp. The **Frame** interface provides specific methods to scan a **CompositeFrame**.

A **CompositeFrame** is made with the same type of **Frame** i.e. the same type of **Keys**. For instance, an **event:data:psa** is a **CompositeFrame** composed only of **data:psa** Frames (**PSAFrames**) while an **event:data** is composed of any kind of AGATA **DataFrame**.

Example:

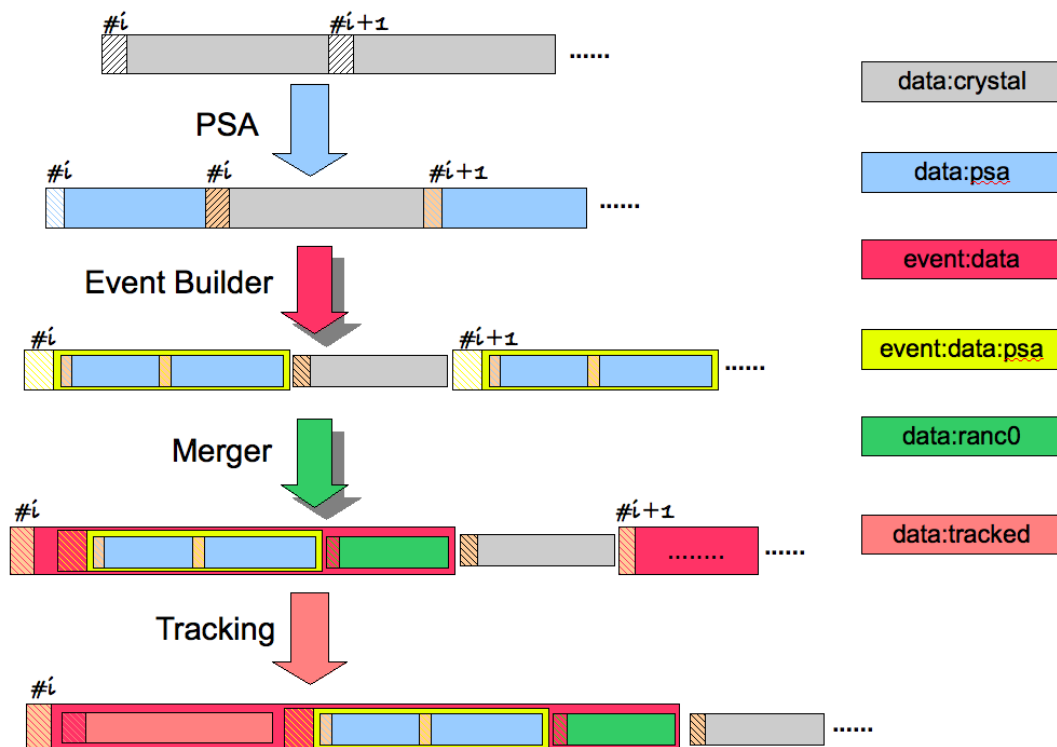


Figure 19: Possible structure of the AGATA Data Flow at different stages

Fig 15 represents a possible structure for the AGATA Data Flow at different stages. At the top level, each crystal is a producer that produces a sequence of **CrystalFrame** (**data:crystal**) with different event number ($\#i$ and $\#i+1$) and/or timestamp. The data flow lines associated to each crystal are processed in parallel by the PSA algorithm that produces, based on the signals, **PSAFrames** (**data:psa**). Depending on the configuration, the input Frame could be consumed or not. In this example, the **CrystalFrame** still exists on the output data flow after the PSA algorithm.

At a global level⁵, the Event Builder merges together all the **PSAFrame** in a **CompositeFrame**

⁵ The definition of the event builder may be different depending on the needs and the version of the associated

(**event:data:psa**) using the event number and/or timestamp. This **CompositeFrame** is packed into another **CompositeFrame** (**event:data**). As well it adds also the **CrystalFrame** that was used to produce the **PSAFrame**. At the merger stage, an ancillary frame (**data:ranc0**) is added to if it has the same event number and/or timestamp. At the tracking stage, the **event:data:psa** frame is used to produce the **TrackedFrame** (**data:tracked**) which is added to the main composite frame. As for the PSA stage, the input frame could be consumed or not. In this example, it is not.

Agata data file (.adf)



An adf file contains a sequence of frames. It should start by a global **ConfigurationFrame** that contains the full definition of the Data Flow (which **Frames** are written, their version number ... etc). The definition of this frame is given in the following sections. Using such a Frame, one can have a system that can adapt itself to the true content of the file and is then auto-configurable. Of course, a global configuration frame could be found anywhere in the file. In this example, the file starts with such a **Frame**, then two **CompositeFrame** (**event:data**) are written followed by a **CrystalFrame** (**data:crystal**).

4.6.3 Interface to Narval

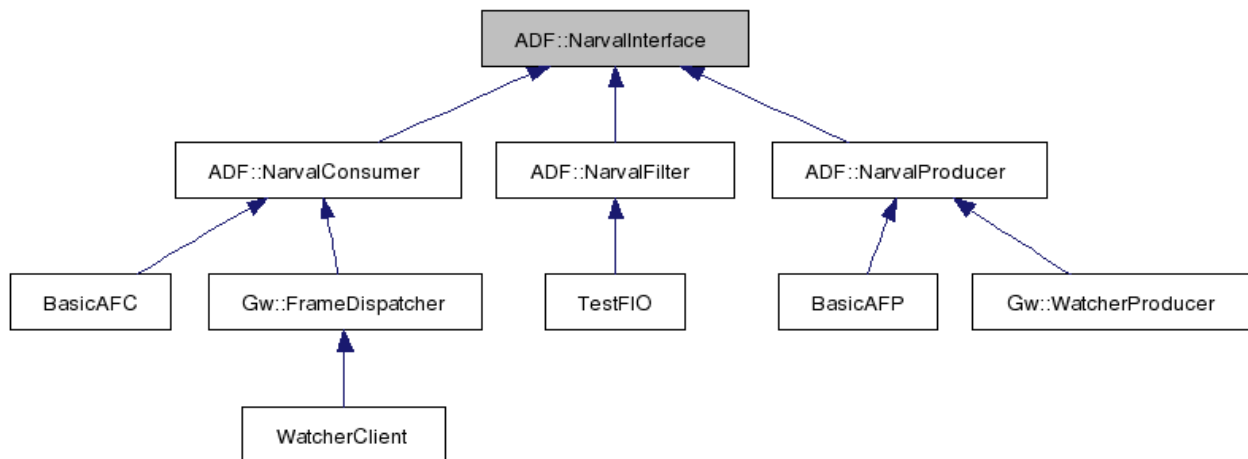


Figure 20 : Some Actors defined in ADF. All inherit from NarvalInterface

In Narval terminology, an algorithm is an actor that processes the Data Flow. In order to help the user to develop a new algorithm, a general interface to Narval called **NarvalInterface** is provided in the ADF module (see Fig. 16). As well, dedicated interfaces to specific actors (**NarvalProducer**, **NarvalFilter** and **NarvalConsumer**) are provided. The interface defines virtual methods the user should implement so that his algorithm could be plugged in a Narval environment. To do that, his class should inherit from one of the three base classes associated to the different type of actors i.e. **NarvalProducer**, **NarvalFilter**, **NarvalConsumer**.

Several actors are already defined in the ADF library (see the inheritance tree Fig. 16). For instance, **BasicAFP** and **BasicAFC** are respectively a producer and a consumer that reads and writes frames from and to regular files. Some forms that summarise the actions of an actor are given in Annex A.

```

class NarvalInterface
{
protected:
    ///! Algo version (used to check suitable frame version ?)
    Version fVersion;
    ///! to send messages to the log server
    LogMessage Log;

public:
...
    ///! To get the ID number for that algorithm
    UInt_t GetPID() const;

    ///! to get from narval a parameter ()
    static Int_t GetGlobalParameter(const Char_t *name, UShort_t &val);
    static Int_t GetGlobalParameter(const Char_t *name, std::string &val);

    static Int_t SetGlobalParameter(const Char_t *name, const UShort_t &);
    static Int_t SetGlobalParameter(const Char_t *name, const std::string &);

    static std::string GetGlobalConfPath();

    ///! To get the algo path associated with the current actor
    const std::string &GetConfPath();

    ///! Have to be overwritten and called in your implementation
    static void process_config(const Char_t *path, UInt_t * error_code, Short_t do_adf_conf = 0);
    ///! To init the internal values (real constructor)
    virtual void process_initialise (UInt_t *error_code) = 0;
    ///! To reset the internal values (real destructor)
    virtual void process_reset (UInt_t *error_code) = 0;
    ///! This method is called every time the daq stops
    virtual void process_stop (UInt_t *error_code) ;
    ///! This method is called every time the daq starts
    virtual void process_start (UInt_t *error_code) ;
    ///! This method is called every time the system pauses data acquisition
    virtual void process_pause (UInt_t *error_code);
    ///! this method is called every time the system resumes data acquisition
    virtual void process_resume (UInt_t *error_code);
};

///! It defines the interface needed to be a consumer.
class NarvalConsumer : public NarvalInterface
{
protected:
    FrameIO &GetFrameIO();
    virtual ConfAgent *GetConfAgent() const;
public:
...
    ///! Have to be overwritten and called in your implementation
    static void process_config (const Char_t*, UInt_t *);
    ///! Ask the algorithm to process the input data block
    virtual UInt_t ProcessBlock (FrameBlock &) = 0;
...
};

///! It defines the interface needed to be a narval actor (Filter).
class NarvalFilter : public NarvalInterface
{
protected:
    FrameIO &GetFrameIO();
    virtual ConfAgent *GetConfAgent() const;
public:
...
    ///! Have to be overwritten and called in your implementation
    static void process_config (const Char_t*, UInt_t *);
    virtual UInt_t ProcessBlock (FrameBlock &, FrameBlock &) = 0;
...
};

///! It defines the interface needed to be a narval actor (producer).
class NarvalProducer : public NarvalInterface
{

```

```
protected:
    FrameIO &GetFrameIO();
    virtual ConfAgent *GetConfAgent() const ;
public:
...
    //! Have to be overwritten and called in your implementation
    static void process_config (const Char_t*, UInt_t *);
    //! Ask the algorithm to process the data block
    virtual UInt_t ProcessBlock (FrameBlock &) = 0;
...
};
```

At running time, Narval calls the configuration method, creates an instance, calls after the initialisation method and fills the algorithm with blocks of data to be treated. At configuration, the path to a directory is given (called `algo_path`). In this directory, the user should put all the files (configuration, calibration etc) needed by the algorithm to run properly. This path is kept in the base class **NarvalInterface** and is thus available to the user by calling the *GetConfPath* method. Other services are provided to know the current structure of the Data Flow (*GetConfAgent*) and to interact with it (*GetFrameIO*). This two aspects are treated in details in the next section.

4.6.4 Interaction of the algorithms with the Data Flow

As seen, the Data Flow is composed of different kind of **Frames** with version numbers. A **Frame** starts with the length of the **Frame**, it is then easy to map it. In ADF, an object is dedicated for that task, it is called a **FrameIO** (see Fig.17). Once a new **Frame** is found, its nature could be checked using the **Key** and in particular the message type. A particular algorithm should be executed only if the suitable input is found. At the initialisation phase, an algorithm registers a **Trigger** to the **FrameIO** object (using the *Register* method). The **Trigger** fully defines what is required for the algorithm to work properly and to produce a new **Frame** that is going to be written on the output data flow. For instance, the PSA algorithm needs a **CrystalFrame** to produce a **PSAFrame**. The **Trigger** for that algorithm could be then defined like this :

```
AgataFrameTrigger *trig = new AgataFrameTrigger("Crystal");

SharedFP *cframe = trig->Add("Agata","data:crystal",ConfAgent::theGlobalAgent(),true);
SharedFP *psaframe = trig->SetOutput("Agata","data:psa",ConfAgent::theGlobalAgent());
```

The first line allocates a new **Trigger** called **Crystal**. The second one is to add to this **Trigger** a condition which is to find on the input data flow a **CrystalFrame**. The third argument is used to know what is the current version number for that type of frame. The forth argument tells the input **Frame** is consumed. Indeed, an algorithm needs as an input a **Frame**. In most of the cases this input **Frame** is used to produce only a new type of **Frame** but in some case one may need also to keep the input frame on the data flow. The third line tells the algorithm produces a **PSAFrame**. Once the required conditions are found on the input data flow, the **FrameIO** object fills the **Frame** and notifies the Actor which then processes the data and fills the output **Frame**. By calling the *Record* method, the **FrameIO** object records the produced **Frame** on the output data flow.

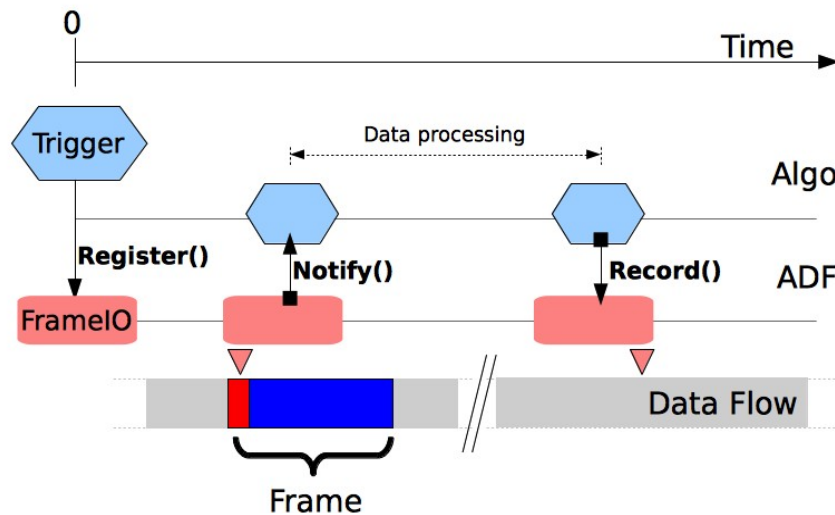


Figure 21: FrameIO and Trigger mechanism

The **Trigger** condition could be more complex. For instance, one can define a **Trigger** on a **CompositeFrame** that contains some **Frames** :

```
AgataFrameTrigger *trigTrack = new AgataFrameTrigger("Tracked");
trigTrack->Add("Agata", "event:data", ConfAgent::theGlobalAgent());
trigTrack->Add("Agata", "data:tracked", ConfAgent::theGlobalAgent());
trigTrack->Add("Agata", "data:ranc0", ConfAgent::theGlobalAgent());
```

In this case, the algorithm is notified only if a composite frame (**event:data**) is found on the input data flow in which there are at least one tracked (**data:tracked**) and a raw ancillary frame **data:ranc0**. For any of the **Frame** defined in the **Trigger**, a flag is attached to determine if the corresponding **Frame** is consumed or not. As it can be guessed, there might be some other frames in the **CompositeFrame** not needed, not known by the trigger. A strategy is required for such a kind of “unknown” frames. The behaviour of the **FrameIO** manager is modified by changing the IO model (see the *SetModel()* method). Three values exist: **kStrict**, **kSafe** and **kGrowing**. Depending of the chosen model, the structure of the output data flow is affected as indicated in the following table.

	<i>Frames defined in Triggers</i>	<i>Frames NOT defined in Triggers</i>
kStrict	Only consumable are consumed	Consumed
kSafe	Only consumable are consumed	Not consumed
kGrowing	Not consumed	Not consumed

To summarize, the **kStrict** mode (default) do exactly what is customized by the users in the trigger definition, keeping only the frames that are not flagged as consumed (undefined frame are consumed whatever they are). In **kGrowing**, nothing is deleted so that the data flow is growing each time it goes through a Filter. The **kSafe** mode is intermediate doing exactly what is customized in the Trigger for the defined frames and keep any undefined frames.

At any time, the full definition of the current data flow is known by the **FrameIO** manager. At the initialisation phase of any algorithm, these informations are read from an ascii file named ADF.conf.

This file should be placed in the directory pointed out by the ADF_CONF_PATH environment variable. Here is a example of such a file:

```
# This is a begin of the ADF configuration record
# Option is Init. It means the Agent is reset before being initialised
#
#####ADF::ConfAgent_beg### Agata 1 Init
#
#
Endian: kLittle
#
# List of factories required
#
# Factories for Keys
#
KeyFactory: Default
KeyFactory: Agata
#
# Factories for Frames
#
FrameFactory: Agata
#
#
# Version of the primary key that structures the data flow
# and the frame expected to reconfigure it
#
PrimaryKey:      Default FS 0 1
#
AutoConf:        Agata conf:global 2 0 Agata conf:global 0 0
#
#
# Current Frames going through the data flow
#
#
Frame: Agata data:tracked 2 0 Agata data:tracked 65000 1
Frame: Agata event:data 2 0 Agata event:data 2 0
Frame: Agata event:data:psa 2 0 Agata event:data:psa 2 0
#
# end of Data Flow Defaults
#
#####ADF::ConfAgent_end###
#
```

All lines starts with a key word, # being used for comments. As it can be seen, it tells the endian type for the data is little endian. In ADF, **Keys** and **Frames** are allocated by **Factories**, thus the name of the required factories are written there. The PrimaryKey tells the length of all the **Frames** is coded using 2 bytes. AutoConf gives the definition of the configuration frame used by ADF to reconfigure itself anytime during the processing. Three different kind of **Frame** (with the associated **Key**) could be found on this particular data flow:

- **data:tracked key version (2,0) and Frame version (65000,1)**
- **event:data key version (2,0) and Frame version (2,0)**
- **event:data:psa key version (2,0) and Frame version (2,0)**

A specific object (**ConfAgent**) keeps this informations so that it is available for the users. In particular, during the definition of the **Trigger**, one may ask for the **ConfAgent** to give the current version number associated to a given Frame (see the third arguments in the Add method).

The content of this ADF.conf file could be packed into a **ConfigurationFrame** and written on the data flow. In particular any .adf file should start with such an autoconf frame (conf:global). It is however not a restriction: a global configuration frame could be written anywhere on the data flow.

This means the current definition of the data flow could change during the processing. The **Frames** exchanged between the **FrameIO** manager and the algorithm are contained (and owned) by the **Trigger**. A global reconfiguration may required to delete and allocate a new Frame. To avoid having

null pointers on the user side, instead of returning frame pointers, the *Add* and *SetOutput* methods of **Trigger** returns shared frame pointers **SharedFP**⁶. Once a non-null **SharedFP** is returned, the user is guaranteed to have a valid pointer even in case of re-configuration. Another aspect is related to the version of the **Frame** an algorithm can handle.

For instance, a particular version of the Tracking algorithm may required, to run properly, the errors associated to the **Hits** provided by the PSA algorithm and thus is likely to run only for some version numbers of the **PSAFrame**. In order to take into account this, a **SharedFP** could be customized thanks to two methods *SetKeyChangeFunction* and *SetFrameChangeFunction*:

```
class SharedFP
{
private:
    Frame *fFrame;
private:
    //! Function used to decide if a key change is acceptable or not
    PF_FactoryItemChange fKeyChange;
    //! Function used to decide if a Frame change is acceptable or not
    PF_FactoryItemChange fFrameChange;
public:
    explicit SharedFP( Frame *f = 0x0 );

    virtual Frame *SetFrame(Frame *);
    virtual Frame *GetFrame() ;

    //! true means this pointer is a zombie (fFrame is null)
    Bool_t IsZombie() const;

    virtual void SetKeyChangeFunction(PF_FactoryItemChange pf);
    virtual void SetFrameChangeFunction(PF_FactoryItemChange pf);
};
```

With this two methods, the user informs the **FrameIO** manager what version number it can handle. At the registration phase or in case of reconfiguration, the **FrameIO** manager checks if the **Trigger** is suitable for the new definition of the data flow. If it is, the **Trigger** is put on the stack of running triggers. If not, it is put on the stack of non-running triggers waiting for a new reconfiguration.

⁶ This is an object that contains a pointer to a Frame

4.7 The ADFE component

This module extends the ADF library by defining objects that need both ADF to access the Data Flow, and in particular the AGATA frames, and ROOT and/or the GammaWare libraries to build for instance histograms or TTree.

4.7.1 Watchers and FrameDispatcher

Triggers are used to exchange **Frames** between different part of a program. Once a **Trigger** has been defined and filled by ADF, the user can process the **Frames** to perform the analysis he wants. In this module is defined a **Watcher**: this is a task that can be associated to a given **Trigger**. For instance, one can define a **Trigger** on a **CrystalFrame** ([data:crystal](#)) and display on screen the different signals or apply some filter to correct the raw signal. All specific **Watchers** should inherit from one of the base classes **Watcher** or **WatcherWithTag** and implement the *Exec* method which contains the treatment anytime the **Trigger** has fired. One of the most common task is to build histograms. In such a case, the histograms are defined in the **Watcher** *constructor* while the are filled in the *Exec* method. However the task perform by the **Watcher** could be also writing the data into another format (ex: ROOT TTree).

```
class Watcher : public TTask
{
protected:
    ///! trigger associated to this watcher
    ADF::DFTrigger *fTrigger;

public:
    Watcher(const char *name, const char *title);

    ///! To set the Frames (through a trigger) associated to this watcher
    virtual Bool_t SetTrigger(ADF::DFTrigger *);
    ///! To know the trigger in which the frame to be watched is embedded
    virtual ADF::DFTrigger *GetTrigger() const;
    ///! watch the current frame ... to be overwritten by the watcher
    virtual void Exec(Option_t *option);
    ///! Display some results
    void ShowCanvas(Option_t *option="") ; /*MENU*
};
```

Facilities exist in the base class like for instance the management of the memory allocated for the spectra. As well, to each **Watcher** is associated a ROOT directory so that the produced histograms can be saved in a ROOT file. A **WatcherWithTag** is a **Watcher** for which several histograms have been tagged. Once a histogram has been tagged, a newly produced histogram could be compared using a Kolmogorov test in order to check whether or not the content of the histograms has been modified. Several **Watchers** are provided in the GammaWare package (see Chapter 5, demos/adf). Of course, a **Watcher** can be used the same way for online and offline analysis.

A **FrameDispatcher** is a consumer in Narval terminology, it receives buffers (from a producer or a filter) to be treated. Moreover, one can add to a **FrameDispatcher** a list of **Triggers** each one being associated to a **Watcher**⁷. The **Watchers** could be associated to the same **Trigger** or different **Triggers**: it is user configurable. The **FrameDispatcher** consumes the input data and once a **Trigger** fired the corresponding **Watcher** is called. As it can be guessed, if several **Watchers** are

⁷ In this sense, it is more a 'trigger' dispatcher.

attached to the dispatcher, the slowest **Watcher** can slow down considerably all the other ones. In order to check things and build a consistent and efficient sequence, histograms are build by the **FrameDispatcher** to get the time required to execute any of the **Watchers**.

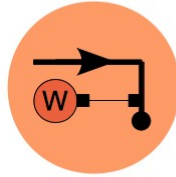


Figure 22: A FrameDispatcher contains a list of Watchers with the associated Trigger

```
class FrameDispatcher : public Watcher, public ADF::NarvalConsumer
{
public:
...
    //! look for a Frame using the Trigger and dispatch it to sustasks
    virtual UInt_t ProcessBlock (ADF::FrameBlock &);

    //! This is the default trigger for any added watcher
    virtual Bool_t SetTrigger(ADF::DFTrigger *);

    //! To set a particular watcher with a particular trigger
    virtual void Add(Watcher *, ADF::DFTrigger *);
...
};
```

Here is a non-exhaustive list of Watchers available in the package for different kind of frames.

- CrystalFrame: **ShowSignals** to display segment and core signals
- PSAFrame: **PSACrystal3D** displays the hits in a single crystal
- TrackedFrame: **Coinc2D** built the Fold distribution, a γ - γ matrix (Doppler corrected) and the spectrum associated to singles.

Chapter 5 - Examples

Chapter 5 - Examples	51
5.1 Some macros in details.....	53
5.2 Some demos in details.....	53
5.2.1 demos/adf.....	53
5.2.1.1 Watchers.....	53
5.2.2 demos/gem.....	59
5.2.3 demos/tools.....	61
5.2.4 demos/physics.....	62
5.2.4.1 GSPlayer.....	62
5.2.4.1 Correlated Space.....	63
5.2.4 demos/tools.....	64

5.1 Some macros in details

5.2 Some demos in details

5.2.1 demos/adf

In this directory you can find examples to show how to interact with the AGATA Data Flow and how to use the other modules of the GammaWare to perform analysis. The directory itself contains different sub-directories (see also the README file) :

- *DefaultWatchers* : it contains all the watchers proposed by default in the Gw package
- *Macros* : it contains ROOT macros to help running applications in ROOT sessions or for some of them as standard compiled programs.
- *Conf* : it contains some configuration files for actors or watchers

5.2.1.1 Watchers

Some default **Watchers** exist, the files are all stored in the *DefaultWatchers* directory. First you need to compile them and load it in a root session (if you would like to use them of course !). To help you doing this, a macro called *LoadWatchers.C* is available in the *Macros* directory. Copy it in the *demos/adf* directory and execute it :

```
root LoadWatchers.C
*****
*                                     *
*   W E L C O M E  t o  R O O T      *
*                                     *
*   Version 5.24/00b  11 October 2009 *
*                                     *
*   You are welcome to visit our Web site *
*   http://root.cern.ch                 *
*                                     *
*****

ROOT 5.24/00b (tags/v5-24-00b@30662, Oct 11 2009, 22:40:07 on macosx64)

CINT/ROOT C/C++ Interpreter version 5.17.00, Dec 21, 2008
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
Processing LoadWatchers.C...
*****
Messages from LoadWatchers.C
*****
- Add /Users/stezow/tmp/share/gw/macros to the macros path
- Add /Users/stezow/tmp/include/gw0.9 to the include path
- Add ./DefaultWatchers to the include path
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/ADFWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/WatcherClient_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/MetaWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/TrackedWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/EventPSAWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/PSAWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/CrystalWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/RancLegnaroWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/EventWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/DanteWatchers_C.so
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/./DefaultWatchers/MyTree_C.so
- Add /path/to/where/are/prisma/includes to the include path

To start online (Local Level) Watchers :
- to watch one selected actor
.x OnlineWatchersLLPC+ or .x OnlineWatchersLLPC+(new TFile("MySpectra.root","UPDATE"))
- to watch simultaneously all the actors
.x OnlineWatchersLLPC+(0x0,true) or .x OnlineWatchersLLPC+(new TFile("MySpectra.root","UPDATE"),true)
```



```

To start online (Global Level) Watchers :
.x OnlineWatchersGLPC+ or .x OnlineWatchersGLPC+(new TFile("MySpectra.root","UPDATE"))
Then start the watching from the ROOT Browser

To start offline Watchers :
1 - Configure OfflineWatcher.C (it could be a copy of of the one in DefaultWatchers)
2 - Run the Watchers with :
.x OfflineWatchers.C+ or .x OfflineWatchers.C+(new TFile("MySpectra.root","UPDATE"))

[Info] ADF_CONF_PATH has been set to /Users/stezow/Soft/Subversion/gammaware/demos/adf
root [1]

```

Basically, it compiles and loads all the default **Watcher** so that you can use them. There are also indications on how to run the **Watchers** in different mode. This is described later in the section. An important point is printed out at the last sentence : the path to ADF_CONF_PATH should be set !

There are two ways to run the **Watchers** : “online” or “offline”. In the “online” mode, the buffer of data are obtained through a socket connection while in the “offline” mode an emulator is used consisting in a producer that reads data from a file and sent them to a consumer which is in the simplest case, a **FrameDispatcher**. Online, the server is any Narval actor. To get the data the name of the machine on which the **Actor** is running as well as the port number (default is 10201) are needed. Fig. 19 shows one of the configuration already that has been used on site ⁸.

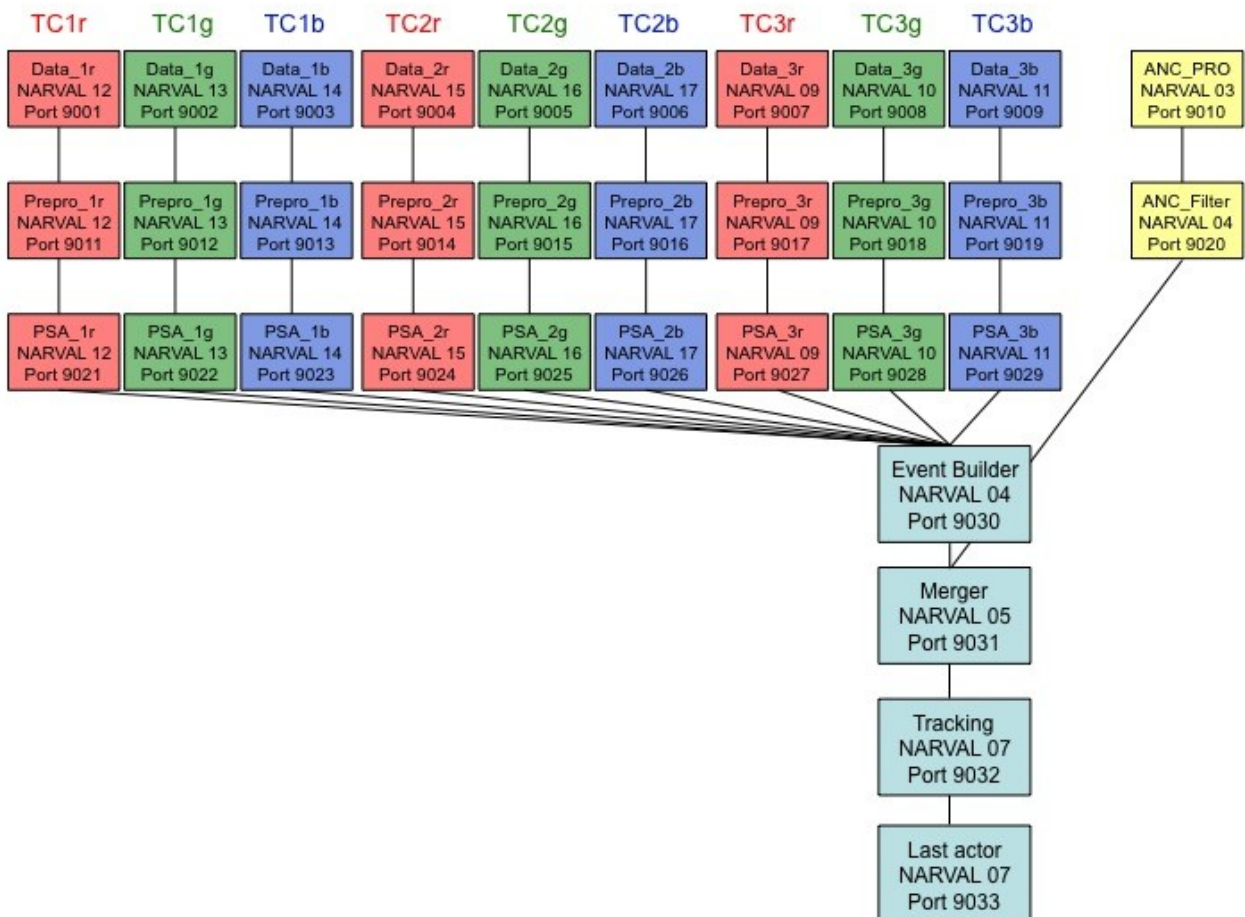


Figure 23: Informations to watch online a topology with three AGATA clusters and an ancillary

⁸ Courtesy of N.Redon

As it has been mentioned, a **Watcher** is a task associated to a given **Trigger**. One thus needs to set up this association. Examples could be found in the macro *Macros/SetupWatchers.C*⁹. This file is used to set up all the default **Watchers** for online watching. Here is a part of this file concerning the Data Flow at the MERGER level i.e. for a data flow composed of **event:data Frames**.

```
Bool_t SetupWatchers(const char *df_type, FrameDispatcher *fd, const char *ext = "") {
...
// MERGER Level //
if ( DFtype.Contains("MERGER") ) {

    AgataFrameTrigger *trig =
        AgataFrameTrigger::Build("Event", "event:data event:data:psa data:ranc1");

    // set the main trigger to coincidences between ancillary et gammas
    fd->SetTrigger(trig);

    fd->Add<TSCoinc>(GetWN("TSCoinc",ext),"TS distribution in events");
    fd->Add<HitsSpectra>(GetWN("HitsSpectra",ext),"Gamma-ray spectra from Hits");

// Any kind of Watcher on event:data:psa, data:ranc1 or
fd->Add<DisplayRancLegnaro>(GetWN("DisplayRanc",ext),
                           "RawData as produced by the Ancillary Filter");
fd->Add<DisplayDante>(GetWN("DisplayDante",ext),
                     "Display Raw and Calibrated Data for Dante");
...
}
```

As it can be seen, a **Trigger** (*AgataFrameTrigger *trig*) is first built. It is named Event and it triggers each time an **event:data Frame** composed of an **event:data:psa Frame** and a **data:ranc0 Frame** is found on the data flow. The **Trigger** is then added as the default trigger to the **FrameDispatcher** (*fd*). To this dispatcher, several **Watchers** (*TSCoinc*, *HistSpectra* ...) are added each time with a name as a first argument and a general title as the second argument. Be careful, in order to avoid potential problems, it is better to have different names for all the **Watchers** in a given session. In a **FrameDispatcher**, each **Watcher** could be associated with different trigger. For that, just add as third argument the pointer to the **Trigger** :

```
fd->Add<TSCoinc>(GetWN("TSCoinc",ext),"TS distribution in events",trig1);
fd->Add<HitsSpectra>(GetWN("HitsSpectra",ext),"Gamma-ray spectra from Hits",trig2);
```

For online analysis, there are two ROOT macros which help you watching the system :

- *OnlineWatchersLLP.C* : Local Level Processing, to watch the data coming out from the initial producers up to just before the Event Builder (or Merger for the ancillary part).
- *OnlineWatchersGLP.C* : Global Level Processing, to watch data from the Event Builder up to the final consumer that dumps data on disk.

These two macros take care of collecting the data from the different actors and sending them to the corresponding **FrameDispatcher** which itself calls all the **Watchers** associated to the **Trigger** that has fired. To start one of the system (here the Local Level Processing Watchers):

```
root -l LoadWatchers.C
...
root [1] .x OnlineWatchersLLP.C+
```

It adds to the task directory all the default **Watchers**. From this menu you can play with the system i.e. start/stop collecting data, see spectra, reset them ... etc.

⁹ Another example is *Macros/DoyTree.C*, explained in details later on

If you have some *adf* files, you can emulate an “online” like system. To do this, in the *demos/adf* directory a macro, *StartServer.C*, is provided to emulate a server. It is a producer that reads data from an adf file (**BasicAFP** producer) and send them on request to the client through a socket. As any **BasicAFP** producer, the path to the files could be configured using a configuration file *BasicAFP.conf* which should be in the working directory (see *Conf* directory for an exemple of such a file).

```
root StartServer.C

...
CINT/ROOT C/C++ Interpreter version 5.17.00, Dec 21, 2008
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
Processing StartServer.C...
*****
Messages from StartServer.C
*****
- Add /Users/stezow/tmp/share/gw/macros to the macros path
- Add /Users/stezow/tmp/include/gw0.9 to the include path
Info in <TUnixSystem::ACLiC>: creating shared library /Users/stezow/Soft/Subversion/gammaware/demos/adf/.WatcherServer_C.so
To start the Server : RunWatcherServer("./")
root [1]
```

A video http://agata.in2p3.fr/VT/Watchers_Default_Online_Offline.mov demonstrates how to start the system. Once it is running you can look at the spectra, change the connection to the server (with the *SetConnection* method in the **FrameDispatcher** menu), tag some spectra ... etc.

Online, the **FrameDispatcher** is associated with a producer that collect the buffers of data from a socket. Of course these data can come from a producer that just read the data from a file (.adf) containing ADF Frames. Without any changes in the **Watcher** you can then use it offline. There is a ROOT macro called *OfflineWatchers.C* in the *Macros* directory that shows you how to proceed. Copy it in the *demos/adf* directory.

```
void OfflineWatchers(TFile *file = 0x0)
{
...
    // chain a producer and a FrameDispatcher to which Watchers are attached
    EmulatorPC<RootAFP,FrameDispatcher> emulator("./");

    if ( ! emulator.Init() ) {
        std::cout << "Cannot Init emulator" << std::endl; return;
    }

    // EB -----
    // output of the EB ... hits only
    MainDir->cd();
    SetupWatchers("EB",emulator.GetConsumer());
    // -----
...
    // run
    emulator.Run();
    // displays statistics and keeps all histograms
    // emulator.GetConsumer()->GetFrameIO().Print();

    // write all the spectra produced by the different watchers in a root file
    MainDir->Write();
}
```

As you can see, it first creates a **Emulator**¹⁰ which consists in a chain of a producer (**RootAFP**) and a consumer (**FrameDispatcher**) to which are attached the **Watchers**. In this case, the function that

10 See Emulator.h in the demos/adf directory

has been used, *SetupWatchers* (defined in *SetupWatchers.C*), is the same that the one used by the online system (*OnlineWatchersGLP.C* and *OnlineWatchersLLP.C*). It illustrates then how to run online/offline the same way. Once the **Watchers** are attached, the **Emulator** is run (*emulator.Run()*) i.e. the data are read from the files defined in *RootAFP.conf*, sent to the dispatcher that called the **Watcher**. In most of the cases, the **Watchers** create spectra that are then saved in the ROOT file if there is one defined (*MainDir->Write()*).

A **Watcher** could be any analysis performed on a given condition i.e. on the input **Frames**. One may need to read some ancillary **Frame**, to perform some calibration then store the produced calibrated data into ROOT TTrees that are extremely performant to analyse multi parametric events. This is quite easy using **Watchers** and already available. Some default **Watchers** produce ROOT TTrees. They are defined into the *DefaultWatchers/MyTree.C* file. TTrees depend strongly on their definition in particular the branches' definitions related to the analysis you would like to perform. Thus, feel free to change/add whatever you need.

→ TO BE WRITTEN :

End of Ttree

How to modify/add a Watcher, with/without additional library.

How to compile differently

How to add the tracking code

→ under this line, up to demos/gem, the text is likely to change a little bit.

In case you need to develop a new **Watcher**, or modify (an existing one) you should customize the macros.

The first one is the one that configure the root session i.e. *LoadWatcher.C*. In particular you should add the lines so that you can add a new library to the system (for ancillaries for instance). A complete example is given in the *demos/adf* directory to deal with Prisma and Dante see also *PrismaAndDanteWatchers.h* and *PrismaAndDanteWatchers.C*.

In the *LoadWatcher.C* macro, you should customise the line **underlined** to match your installation.

```
// ----> ANCILLARY_PART ////////////////////////////////////
// Directory where are the includes of the ancillary part
tmp = "/path/to/where/are/prisma/includes"; printf("- Add %s to the include path \n",tmp.Data());
tmp.Prepnd(" .include ");
gROOT->ProcessLine(tmp.Data());

// ----> Additional libraries if required (ANCILLARY PART)
gSystem->Load("libPrisma");

// ----> Compile and load the watchers for ancillaries
gROOT->ProcessLine(".L PrismaWatchers.C+");

// ----> ANCILLARY_PART ////////////////////////////////////
```

The first modification consists in giving the path to the includes for additional package if it is required to compile the additional **Watchers**. The second modification consists in loading the additional libraries and in this case the prisma library. The path to the prisma library should be set. This is done by calling the *preproc.csh* script provided in the prisma package. The last one compiles and load the Prisma and Dante **Watchers** contained in the *PrismaWatchers.C* macro.

To setup the system so that it could run the new **Watchers**, copy the macro *SetMyWatchers.D* into *SetMyWatchers.C* and modify it to make the link between the **Watchers** you would like to run and

the associated Triggers. Ex :

```
#ifndef __CINT__
#include "SetupMyWatchers.h"

// include's required by your watchers
#include "MetaWatchers.h"
#include "PrismaAndDanteWatchers.h"

using namespace Gw;
using namespace ADF;

Bool_t SetupMyWatchers(const char *df_type, Gw::FrameDispatcher *fd)
{
    Bool_t ok = false; TString DFtype = df_type;

    // here define your watchers and their trigger ... see SetupWatchers.C for example
    // ranc1 containing Prisma and Dante data
    if ( DFtype.Contains("ranc1gamma") ) {
        // The trigger is on an event frame composed of an ancillary and an tracked frame
        AgataFrameTrigger *trig = new AgataFrameTrigger("MyAncAndTracked");
        trig->Add("Agata", "event:data", ConfAgent::theGlobalAgent());
        trig->Add("Agata", "data:tracked", ConfAgent::theGlobalAgent());
        trig->Add("Agata", "data:ranc1", ConfAgent::theGlobalAgent());
        fd->SetTrigger(trig);

        // compute from the ancillary Frame a VertexFrame available using the static methods of
        VertexWatcher
        fd->GetDirectory()->cd(); Watcher *vertex = new AncVertex("Source", "To set a Global Vertex
from Ancillary");
        fd->Add( vertex );
        // from the static method of VertexWatcher, computes the beta distribution
        fd->GetDirectory()->cd(); Watcher *dopplercorrector = new GlobalDC("Recoil", "Recoil velocity
distribution");
        fd->Add( dopplercorrector );
        // build a tree for ancillary data
        fd->GetDirectory()->cd();
        Watcher *ancntree = new DoMyAncillaryTree("AncTree", "MyTree for ancillaries in coincidence with
tracked");
        fd->Add( ancntree );
        // build a tree for gamma data
        fd->GetDirectory()->cd();
        Watcher *gamtree = new DoMyGammaTree("GTree", "MyTree for gamma in coincidence with
ancillary");
        fd->Add( gamtree );

        ok = true;
    }
    return ok;
}
```

The last two Watchers (DoMyAncillaryTree and DoMyGammaTree) built TTree in different ROOT files. Because they have a common trigger, the entry number is consistent in the two TTree. As they are saved in separate ROOT files you can analyse them separately but due to the consistency in the entry number, you can also use the ROOT TTree friend facility to perform analysis that required coincidences between ancillaries and gammas.

5.2.2 demos/gem

Go to the gem sub-directory and start root, you will get the following message:

```

root
*****
Messages from rootlogon.C
*****
- Try to load some needed libraries
libGWCORE loaded
libGWPHYSICS loaded
libGWGEM loaded
root[0]

```

A level scheme object could be declared and built by exporting the definition from an ags file (Radware <http://radware.phy.ornl.gov/> format for level schemes):

```

root[0] Gw::LevelScheme lev
root[1] lev.Import("simul.ags","dummy")

```

A summary of the content is given on the command line:

```

---> Reading file simul.ags
    24 levels, 24 gammas and 3 bands supposed to be in this level scheme
    24 levels correctly read in 3 bands
    3 bands correctly read
    24 gammas correctly read
... Reading file simul.ags DONE
(Int_t)1

```

As for any ROOT object, the level scheme can be drawn with the following command:

```

root[1] lev.Draw()

```

It can be seen in the Doxygen documentation that a LevelScheme is a an object composed of Levels (NuclearLevels), Links (GammaLink, XGammaLink etc) and Cascades. A Cascade is a particular decay path inside the level scheme.

Another possibilty offered by the GammaWare is to select randomly cascades of -rays in a level scheme. Open in your favorite editor the ensdfdemo.C macro or have a look at it through the doxygen documentation on the Agata Data Analysis web site <http://agata.in2p3.fr> in the macros directory. Two functions are defined. One of them is reported in the following lines:

```

00030 void oneLS(const Char_t *filename, const Char_t *nucleus, Int_t nbcas = 100)
00031 {
00032 // load the AGS file
00033 BaseGEM random;
00034 if ( random.Import(filename,nucleus) == 0 ) {
00035 cout << " Level Scheme not correctly loaded " << endl; return;
00036 }
00037
00038 // some spectra to be filled
00039 TH1F *K = new TH1F("K","K",100,0,100);
00040 TH1F *h = new TH1F("ProjTot","ProjTot",8192,0,4096);

```

```

00041
00042 // start the Monte-Carlo
00043 Cascade cas; GammaLink *gam;
00044 for (Int_t i = 0; i < nbcas; i++ ) {
00045
00046     random.DoCascade(cas);
00047
00048     K->Fill(cas.GetSize() );
00049     for (Int_t j = 0; j < cas.GetSize(); j++ ) {
00050         gam = (GammaLink *)cas.At(j); h->Fill(gam->GetEnergy().Get());
00051     } // j
00052 } //i
00053 h->Draw();
00054 }

```

The function name is `oneLS` and it illustrates how cascades are randomly generated from a Radware (.ags) or ENSDF (.ens) level scheme. It takes three arguments: a filename (in which is stored the level scheme), the nucleus to which corresponds the level scheme and the number of cascades to generate. A BaseGEM object is created (line 33) and initiated (line 34). In case of success, two histograms are allocated for the multiplicity and the -ray spectrum. A Cascade is filled in the loop that starts line 44. As you can see, the BaseGEM object (called `random`) owns a method (`DoCascade`) that fills randomly the cascade given as an argument. After this call, the multiplicity of the cascade is obtained through the `GetSize()` method (line 48) and each individual -rays composing the cascade could be obtained as it is shown line 50 (`At(j)` method). Just after, the energy of the GammaLink is incremented in the histogram which is drawn when the loop stops (`h->Draw()`). To run this function, go to the `demos/gem` directory, start root and type the following line:

```

root[0] .L ensdfdemo.C
root[1] oneLS("simul.ags","dummy",1000)

```

The first line load the macros. Once it is done, the different functions can be called on the command line. In the second line the `oneLS` function is called and the simulated spectra will appear on your screen.

5.2.3 demos/tools

ROOT works with its own format for histograms (1D, 2D and even 3D). However it might be convenient to export such spectra to analyse them within another framework. As well it is important to be able to read, in the forthcoming years, spectra that were created few years ago and sometime with programs that don't exist anymore. The `HistoDB` class provides an interface to store/load spectra to/from different frameworks. It has been designed so that new possibilities could be added without having to recompile the GammaWare. Go to the `demos/tools` directory, start a root session and build a new Database System for Histograms (`Gw::HistoDB`).

```

root[0] Gw::HistoDB db

```

For the moment, it is connected to any database system. What is required, for instance, is to extract a spectrum from a `gpsi` file. To do that, you must connect `db` with the file in which are written the

spectra (in this case run2.s) by calling the following method:

```
root[1] db.Open("archive","gpsi:run2.s")
```

For the database service, “archive” is the name to identify a source of spectra that is of gpsi type and stored in the file run2.s. To see the content of the file:

```
root[2] db.ls()
```

You will see the name of the 242 spectra written in the file. To read the one named “P3#11-E”, you should create a ROOT histogram (TH1F), give it the name of the spectra you want to load in memory and call the import operator (>>):

```
root[3] TH1F h
root[4] h.SetName("P3#11-E")
root[5] db >> h
root[6] h.Draw()
```

Another possibility is to load the spectrum by its number in the database system, in our case #216:

```
root[7] TH1 *sp216 = db.Get(216)
```

As well, you may need only spectra with their name finishing by “-C”. It can be done in a single operation by using a THStack object:

```
root[8] THStack stack
root[9] stack.SetName("*-C")
root[10] db >> stack
```

The stack is now filled only with the right histograms as it can be seen by listing the content with stack.ls(). The loaded histograms can be saved in ROOT files but they cannot be stored in another gpsi file because this program is not maintained anymore. Thus, the export methods are not implemented in the HistoDB interface. However, it could be useful to export histograms so that they can be read, for instance, by gf3 which is the Radware program to interactively work on spectra. To do this, you must connect the database service to a Radware database. Because a spectrum is stored in a single file (.spe), a database in this case is a single directory. So to open a connection with the current directory:

```
root[11] db.Open("torad","rad:${PWD}")
```

Two databases are now open and you can switch from one to the other one by using the database's name:

```
root[12] db << "archive"; // switch to gpsi
root[13] db << "torad"; // switch to rad
```

To convert the spectrum named Tot-sum from gpsi to Radware:

```
root[14] TH1F sp; sp.SetName("Tot-sum"); db << "archive" >> sp << "torad" << sp;
```

You will see that the file Tot-sum.spe has been written in your current directory. As well the THStack could be used instead of a single histogram so that many of them could be converted at the same time.

5.2.4 demos/physics

5.2.4.1 GSPlayer

There is a macro that demonstrates how to connect a spectrum to the GSPlayer. First create a canvas:

```
root [1] TCanvas* c1 = new TCanvas("c1","titre",10,10,700,500);
```

Draw a spectrum with the help of the HistoDB class:

```
root [2] Gw::HistoDB db;
....
root [7] h->SetAxisRange(2150, 2250);
root [8] h->Draw();
```

To connect the GSPlayer to the spectrum, just create the GSPlayer object:

```
root [9] Gw::GSPlayer* s = new Gw::GSPlayer();
```

And connect to the canvas:

```
root [10] s->Connect(c1);
```

The GSPlayer is now connected to the canvas. One can place the mouse on the top of the peak to draw a Peak1D object by pressing key 'p'. This Peak1D object will automatically adjust onto the spectrum peak.

One can run the fit procedure by the command:

```
root [2] s->FitAll();
```

All the peaks will be fitted default wise with a gaussian shape function with a linear background. The results are stored in the Peak1D objects and can be printed out:

```
root [14] s->Print();
OBJ: Gw::Peak1D Peak1  Gw::Peak1D
Height: 10207.1
Intensity: 65114.4
Position: 2172.4 FWHM: 5.0
Bkg left1: 2154.1 Bkg left2: 2160.1 Bkg right1: 2184.1 Bkg right1: 2190.1
....
OBJ: Gw::Peak1D Peak3  Gw::Peak1D
Height: 13087.2
Intensity: 72924.4
Position: 2224.8 FWHM: 5.2
Bkg left1: 2207.4 Bkg left2: 2213.4 Bkg right1: 2237.4 Bkg right1: 2243.4
```

The fitting method could be called through a context menu with pressing key 'f'.

5.2.4.1 Correlated Space

There are three macros two for writing (from a **TH2** file and randomly) a two dimensional symmetrical correlated space tree and one for reading back with using a gating procedure. We will present in details two of them.

To write a correlated space from a TH2 spectrum stored in a Root file. First retrieve the spectrum:

```
root [1] TFile* fileIn = new TFile("Eu152-gg-mat.root", "READ");
root [2] TH2F* h2 = (TH2F*)fileIn->Get("HEGamma_EGamma");
```

Then open the root file where one will stored the correlated space:

```
root [3] TFile* fileOut = new TFile("test1.root", "RECREATE");
root [4] fileOut->cd();
```

Create the specific correlated space pointer for tree of integers and fill the tree:

```
root [5] Gw::CorrelatedSpaceTree2I* coor = new Gw::CorrelatedSpaceTree2I();
root [6] coor->FillFromH2(h2);
```

Write and close the root files:

```
root [7] coor->Write();
root [8] fileOut->Close();
root [9] fileIn->Close();
```

Once the tree is written one can read it back with a filter. First open the root file and retrieve the correlated space object:

```
root [1] TFile* fileIn = new TFile("test1.root", "READ");
root [2] Gw::CorrelatedSpaceTree2I* coor =
      (Gw::CorrelatedSpaceTree2I*)fileIn->Get("Correlated_Space");
```

Create the filter object and optionally set its name and add it in a root folder:

```
root [3] Gw::CubicFilter* filter = new Gw::CubicFilter();
root [4] filter->SetName("Gate0");
root [5] TFolder* folder = gROOT->GetRootFolder()->AddFolder("Filter", "Folder");
root [6] folder->Add(filter);
```

Add the gates and make the projection with the given filter:

```
root [7] filter->AddGate(2815, 2824);
root [8] coor->Project(filter);
```

Retrieve the projected histogram and draw it:

```
root [9] TH1D* h = coor->GetHisto();
root[10] h->Draw();
```

5.2.4 demos/tools

There is an macro that shows how to fill a Agata container from gamma ray events generated with the Gem module.

Instantiated the Agata event display and get the Agata container pointer:

```
00046 void AgataDisplay(Int_t nEvents = 100)
00047 {
....
00050 Gw::AgataEventDisplay* dis = Gw::AgataEventDisplay::Instance();
00051 Gw::AgataEventContainer* agataContainer = dis->GetEventContainer();
```

One can set the track style Cone ou Rectangle, the tracks will respectively be drawn as cone or as boxes, the width depending on the energy of the hit associated to the tracklet:

```
00052 dis->SetTrackStyle("Cone");
```

The next part belongs to the GEM part (see for details upper section):

```
00055 Gw::GEM gem;
00057 Gw::BaseGEM *random = 0x0;
00058 random = new Gw::BaseGEM();
....
```

Starts the event loop:

```
00065 for (Int_t i = 0; i < nEvents; ++i) {
00068 random->DoCascade(cas,"");
....
```

Loop over the number of gamma links:

```
00070 for (Int_t j = 0; j < cas.GetSize(); j++) {
00072 gam = (Gw::GammaLink *)cas.At(j);
00073 Double_t e = gRandom->Gaus(gam->GetEnergy().Get(),1);
....
```

Compute randomly the associated hits. Get a new track to be filled and set the first hit:

```
00086 Gw::TrackHit* hit = agataContainer->NewTrackHit();
00087 hit->SetX(x/metricConv);
00088 hit->SetY(y/metricConv);
00089 hit->SetZ(z/metricConv);
00090 hit->SetE(e);
....
```

Set the also the first hit in hit list associated to the track:

```
00093 Gw::StdHit* shit = hit->NewHit();
00094 shit->SetX(x/metricConv);
```

```
00095      shit->SetY(y/metricConv);
00097      shit->SetE(e*0.6);
....
```

Compute a secondary hit for the track in the Agata detector:

```
00099      Int_t ndiff = Int_t(gRandom->Uniform(0, 1) + 0.5);
00100      for (Int_t k = 0; k < ndiff; ++k) {
00101          shit = hit->NewHit();
....
00108          shit->SetXYZ(x/metricConv,y/metricConv,z/metricConv);
00109          shit->SetE(e*0.4/(ndiff+1) );
00110      }
00111  }
....
```

Fill the track in the container for Agata:

```
00113      agataContainer->FillTracks("Agata");
....
```

Ended the event loop. Load the geometry and open the event display:

```
00120  dis->AddGeometry(Gw::AgataGeometryTransformer::ImportAgata(...);
00121  dis->ShowDisplay();
```

Glossary

To be written

Actor

Narval

Consumer

Producer

Filter

Annex A : summary form for some actors

--> General Informations

Name: **BasicAFC**
Type: Consumer
Version: Follow ADF
Contact person: Olivier Stezowski

Web sites (package, documentation, svn etc ...):
to get the code from the svn server
 svn checkout svn+ssh://anonsvn@anonsvn.in2p3.fr/agata/gammaware/trunk/src/adf adf
to browse the svn server (username,id: Agata AgataSoftware)
 https://cvs.in2p3.fr/agata/
html documentation of the code
 http://www.ipnl.in2p3.fr/gammaware/doc/html/

Description of the algorithm:
 An consumer that dumps all frames in a chain of files (size of one file restricted to 2Gb).
 It take care to write at the beginning of each new file a global ConfigurationFrame

Dependencies (to other libraries): ADF Only

--> Parametrisation of the Algorithm

Input files needed (global initialisation):
 BasicAFC.conf to be placed in algo_path
 (not mandatory: there are default but probably not suitable for online processing!)

Algorithm parameters (to modify the way the algorithm works):
 None

Parameters to check the running state (published):
 Only informations send to the log collector

Spectra produced (published):
 None

--> Interaction with the Agata Data Flow and other Actors

Input Data Frames:
 Any kind

Produced Data Frame:
 None

Meta informations to be shared with other algorithms:
 ADF.conf

--> General Informations

Name: **BasicAFP**
Type: Producer
Version: Follow ADF
Contact person: Olivier Stezowski

Web sites (package, documentation, svn etc ...):
to get the code from the svn server
 svn checkout svn+ssh://anonsvn@anonsvn.in2p3.fr/agata/gammaware/trunk/src/adf adf
to browse the svn server (username,id: Agata AgataSoftware)
 https://cvs.in2p3.fr/agata/
html documentation of the code
 http://www.ipnl.in2p3.fr/gammaware/doc/html/

Description of the algorithm:
 It read Frames from a chain of files (ex: BaseName0.adf BaseName1.adf ...).
 It takes care of the global ConfigurationFrame find at the beginning of each new file

Dependencies (to other libraries): ADF Only

--> Parametrisation of the Algorithm

Input files needed (global initialisation):

BasicAFC.conf to be placed in algo_path
(not mandatory: there are default but probably not suitable for online processing!)

Algorithm parameters (to modify the way the algorithm works):

None

Parameters to check the running state (published):

Only informations send to the log collector

Spectra produced (published):

None

--> Interaction with the Agata Data Flow and other Actors

Input Data Frames:

Any kind

Produced Data Frame:

NONE

Meta informations to be shared with other algorithms:

ADF.conf

Annex B : Frame's definitions

Note : version numbers starting with 65000 are reserved for development versions.

Agata - data:crystal – Version(0,0)				
ADFObject	Methods	Data name	In memory	In frame
Global	LinkItems(), Get/SetItems() or Get/SetUID(), Get/SetStatus()	CrystalID	UShort_t	UShort_t
		CrystalStatus	UShort_t	UShort_t
GeSegment	GetSegment(0) +Get/SetStatus() +Get/SetID() +Get/SetE()	Length Status ID Energy Trace	Int_t UShort_t UShort_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t UShort_t[]
...
GeSegment	GetSegment(35) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/Set()/[]	Length Status ID Energy Trace	UInt_t UShort_t UShort_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t UShort_t[]
GeCore	GetCore(0) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/SetT() +Get/Set()/[]	Length Status ID Energy Time Trace	UInt_t UShort_t UShort_t Double_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t Float_t UShort_t[]
GeCore	GetCore(1) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/SetT() +Get/Set()/[]	Length Status ID Energy Time Trace	UInt_t UShort_t UShort_t Double_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t Float_t UShort_t[]

Agata - data:ccrystal – Version(0,0)				
ADFObject	Methods	Data name	In memory	In frame
Global	LinkItems(), Get/SetItems() or Get/SetUID(), Get/SetStatus()	CrystalID	UShort_t	UShort_t
		CrystalStatus	UShort_t	UShort_t
GeSegment	GetSegment(0) +Get/SetStatus() +Get/SetID() +Get/SetE()	Length Status ID Energy Trace	Int_t UShort_t UShort_t Double_t Float_t[]	Int_t UShort_t UShort_t Float_t Float_t[]
...
GeSegment	GetSegment(35) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/Set()/[]	Length Status ID Energy Trace	UInt_t UShort_t UShort_t Double_t Float_t[]	Int_t UShort_t UShort_t Float_t Float_t[]
GeCore	GetCore(0) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/SetT() +Get/Set()/[]	Length Status ID Energy Time Trace	UInt_t UShort_t UShort_t Double_t Double_t Float_t[]	Int_t UShort_t UShort_t Float_t Float_t Float_t[]
GeCore	GetCore(1) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/SetT() +Get/Set()/[]	Length Status ID Energy Time Trace	UInt_t UShort_t UShort_t Double_t Double_t Float_t[]	Int_t UShort_t UShort_t Float_t Float_t Float_t[]

Agata - data:crystal – Version(65000,0)				
ADFObjct	Methods	Data name	In memory	In frame
Global	LinkItems(), Get/SetItems() or Get/SetUID(), Get/SetStatus()	CrystalID	UShort_t	UShort_t
		CrystalStatus	UShort_t	UShort_t
GeSegment	GetSegment(0) +Get/SetStatus() +Get/SetID() +Get/SetE()	Length Status ID Energy Trace	Int_t UShort_t UShort_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t UShort_t[]
...
GeSegment	GetSegment(35) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/Set()/[]	Length Status ID Energy Trace	UInt_t UShort_t UShort_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t UShort_t[]
GeCore	GetCore(0) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/SetT() +Get/Set()/[]	Length Status ID Energy Time Trace	UInt_t UShort_t UShort_t Double_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t Float_t UShort_t[]
GeCore	GetCore(1) +Get/SetStatus() +Get/SetID() +Get/SetE() +Get/SetT() +Get/Set()/[]	Length Status ID Energy Time Trace	UInt_t UShort_t UShort_t Double_t Double_t UShort_t[]	Int_t UShort_t UShort_t Float_t Float_t UShort_t[]

Agata - data:psa – Version(0,0)				
ADFOject	Methods	Data name	In memory	In frame
Global	LinkItems() Get/SetItems()	CrystalID	UShort_t	UShort_t
		CrystalStatus	UShort_t	UShort_t
		CoreE0	Float_t	Float_t
		CoreE1	Float_t	Float_t
		CoreT0	Float_t	Float_t
		CoreT1	Float_t	Float_t
		PSAStatus	UShort_t	UShort_t
	GetNbHit()	NbHit	UShort_t	UShort_t
PSAHit	GetHit(0)/NewHit() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT()	Status	UShort_t	UShort_t
		X	Double_t	Double_t
		Y	Double_t	Double_t
		Z	Double_t	Double_t
		E	Double_t	Double_t
		T	Double_t	Double_t
...
PSAHit	GetHit(GetNbHit())/NewHit() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT()	Status	UShort_t	UShort_t
		X	Double_t	Double_t
		Y	Double_t	Double_t
		Z	Double_t	Double_t
		E	Double_t	Double_t
		T	Double_t	Double_t

Agata - data:psa – Version(65000,0)				
ADFObjct	Methods	Data name	In memory	In frame
Global	LinkItems() Get/SetItems()	CrystalID	UShort_t	UShort_t
		CrystalStatus	UShort_t	UShort_t
		CoreE1	Float_t	Float_t
		CoreE2	Float_t	Float_t
		T0	Short_t	Short_t
		PSAStatus	UShort_t	UShort_t
	GetNbHit()	NbHit	UShort_t	UShort_t
PSAHit	GetHit(0)/NewHit() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT() +Get/SetDX() +Get/SetDY() +Get/SetDZ() +Get/SetDE() +Get/SetDT()	Status	UShort_t	UShort_t
		X	Double_t	Double_t
		Y	Double_t	Double_t
		Z	Double_t	Double_t
		E	Double_t	Double_t
		T	Double_t	Double_t
		DX	Double_t	Double_t
		DY	Double_t	Double_t
		DZ	Double_t	Double_t
		DE	Double_t	Double_t
		DT	Double_t	Double_t
...
PSAHit	GetHit(GetNbHit())/NewHit() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT() +Get/SetDX() +Get/SetDY() +Get/SetDZ() +Get/SetDE() +Get/SetDT()	Status	UShort_t	UShort_t
		X	Double_t	Double_t
		Y	Double_t	Double_t
		Z	Double_t	Double_t
		E	Double_t	Double_t
		T	Double_t	Double_t
		DX	Double_t	Double_t
		DY	Double_t	Double_t
		DZ	Double_t	Double_t
		DE	Double_t	Double_t
		DT	Double_t	Double_t

Agata - data:psa – Version(65000,1)				
ADFObjct	Methods	Data name	In memory	In frame
Global	LinkItems() Get/SetItems()	Anonymous	UShort_t + ...	UShort_t + ...
		CrystalID	UShort_t	UShort_t
		CrystalStatus	UShort_t	UShort_t
		CoreE0	Float_t	Float_t
		CoreE1	Float_t	Float_t
		CoreT0	Float_t	Float_t
		CoreT1	Float_t	Float_t
		PSAStatus	UShort_t	UShort_t
		Spare	UShort_t	UShort_t
	GetNbHit()	NbHit	UShort_t	UShort_t
PSAHit	GetHit(0)/NewHit() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT() +Get/SetDX() +Get/SetDY() +Get/SetDZ() +Get/SetDE() +Get/SetDT()	Status	UShort_t	UShort_t
		X	Double_t	Double_t
		Y	Double_t	Double_t
		Z	Double_t	Double_t
		E	Double_t	Double_t
		T	Double_t	Double_t
		DX	Double_t	Double_t
		DY	Double_t	Double_t
		DZ	Double_t	Double_t
		DE	Double_t	Double_t
		DT	Double_t	Double_t
...
PSAHit	GetHit(GetNbHit())/NewHit() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT() +Get/SetDX() +Get/SetDY() +Get/SetDZ() +Get/SetDE() +Get/SetDT()	Status	UShort_t	UShort_t
		X	Double_t	Double_t
		Y	Double_t	Double_t
		Z	Double_t	Double_t
		E	Double_t	Double_t
		T	Double_t	Double_t
		DX	Double_t	Double_t
		DY	Double_t	Double_t
		DZ	Double_t	Double_t
		DE	Double_t	Double_t
		DT	Double_t	Double_t

Agata - data:tracked – Version(0,0)				
ADFOject	Methods	Data name	In memory	In frame
	GetNbGamma()	NbGamma	UShort_t	UShort_t
TrackedHit	GetGamma(0)/NewGamma() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT()	Status X Y Z X1 Y1 Z1 E T	UShort_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t	UShort_t Float_t Float_t Float_t Float_t Float_t Float_t Float_t Float_t
...
TrackedHit	GetGamma(GetNbGamma()) /NewGamma() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT()	Status X Y Z X1 Y1 Z1 E T	UShort_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t	UShort_t Float_t Float_t Float_t Float_t Float_t Float_t Float_t Float_t

Agata - data:tracked – Version(65000,0)				
ADFObjct	Methods	Data name	In memory	In frame
	GetNbGamma()	NbGamma	UShort_t	UShort_t
TrackedHit	GetGamma(0)/NewGamma() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT() +Get/SetDX() +Get/SetDY() +Get/SetDZ() +Get/SetDE() +Get/SetDT()	Status X Y Z X1 Y1 Z1 E T DX DY DZ DX1 DY1 DZ1 DE DT	UShort_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t	UShort_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t
...
TrackedHit	GetGamma(GetNbGamma()) /NewGamma() +Get/SetX() +Get/SetY() +Get/SetZ() +Get/SetE() +Get/SetT() +Get/SetDX() +Get/SetDY() +Get/SetDZ() +Get/SetDE() +Get/SetDT()	Status X Y Z X1 Y1 Z1 E T DX DY DZ DX1 DY1 DZ1 DE DT	UShort_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t	UShort_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t Double_t